INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in “sectioning” the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
DE FANTI, Thomas Albert, 1948-
THE GRAPHICS SYMBIOSIS SYSTEM--AN INTERACTIVE MINI-COMPUTER ANIMATION GRAPHICS LANGUAGE DESIGNED FOR HABITABILITY AND EXTENSIBILITY.

The Ohio State University, Ph.D., 1973
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

© 1973
THOMAS ALBERT DE FANTI
ALL RIGHTS RESERVED
THE GRAPHICS SYMBIOSIS SYSTEM—
AN INTERACTIVE MINI-COMPUTER ANIMATION GRAPHICS LANGUAGE
DESIGNED FOR HABITABILITY AND EXTENSIBILITY

DISSERTATION
Presented in Partial Fulfillment of the Requirements
for the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Thomas Albert De Fanti, B.A., M.S.

The Ohio State University
1973

Approved by

Lee J. White
Adviser
Department of Computer and Information Science
Professors Lee J. White, Larry H. Reeker, Balakrishnan Chandrasekaran and Charles A. Csuri served as advisers for the dissertation and constituted the reading committee. Much of the motivation to thoroughly implement the system was provided directly or indirectly by these gentlemen. Professor White deserves special thanks for reading and re-reading the dissertation as often as he did.

Professor Charles Csuri provided direction to the project and conscientiously acted as system resident artist to create data and critically evaluate the habitability of the system with respect to his colleagues both in Art and Education.

Manfred Knemeyer was responsible for choosing the equipment, maintaining the PDP-11 Monitor, and writing various I/O drivers. He also wrote the matrix multiplication routines and the basic algorithms for rotation and segmentation, and was often called upon for insights into the problems and solutions faced while developing the system. He was responsible for the algorithms behind shading and linear interpolation as they now exist.

Gerard Moersdorf is to be credited with the implementation
of "TEXT", the debugging technique, and with rewriting all the disk-oriented code. He recently developed the method for swapping assembler modules. Gerry helped fill in the code and thus bring the system to fruition much faster than could have been done otherwise. Some of the system documentation is due to him, as well as most of the efforts to make the system compatible with other similar systems. Gerry has written most of the system utility macros as well.

Mark Gillenson is the system's sophisticated user. Pursuing his own project in Artificial Intelligence and Pattern Recognition, he has coded the shading algorithms, and authored the "FILL", "WARP", and "INTERP" commands. Much of the system's visual quality is due to Mark's work and he is to be credited with many months of work that made the system usable that much sooner.

The implementation of the Graphics Symbiosis System has been a group effort by the members of the Computer Graphics Research Group at The Ohio State University, supported in part by National Science Foundation Grant GJ-204.

All photographs are reproduced with the permission of The Ohio State University Research Foundation.
VITA

1969........B.A., Queens College, New York City
1969-1970.....University Fellow, The Ohio State University, Columbus Ohio
1970.........M.S., The Ohio State University, Columbus, Ohio
1970-1971.....Research Associate responsible for the PDP-10 System of the Department of Computer and Information Science, The Ohio State University, Columbus, Ohio
1971-1972.....University Fellow, The Ohio State University, Columbus, Ohio
1972-1973.....Research Associate, The Computer Graphics Research Group, The Ohio State University, Columbus, Ohio

PUBLICATIONS

FIELDS OF STUDY

Undergraduate...

Mathematics: Professor L. Mansfield
Romance Languages: Professor R. Picken

Graduate...

Programming language Theory: Professor L. White
and Professor L. Reeker
Computational Linguistics: Professor L. Reeker
Automata Theory: Professor L. Reeker
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>VITA</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>FRONTISPICE</td>
<td>viii</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
</tbody>
</table>

## Chapter

1. HABITABILITY IN A GRAPHICS SYSTEM .................................. 24
2. EXTENSIBILITY IN A GRAPHICS SYSTEM ................................. 66
3. IMPLEMENTATION DETAILS .................................................... 93
4. THE FUTURE ........................................................................ 145
5. DISCUSSION OF THE IMPACT ON COMPUTER GRAPHICS AND SOCIETY .... 162

REFERENCES .............................................................................. 169

APPENDIX

A. SYNTAX OF THE LANGUAGE ..................................................... 172
B. SYSTEM CONFIGURATION ....................................................... 192
C. PROGRAM EXAMPLES ............................................................ 193
LIST OF FIGURES

Frontispiece ................................................. viii
Figure 1—Environment ........................................ 30
Figure 2—Devices ............................................. 31
Figure 3—Scenario ............................................ 49
Figure 4—Setintensity ....................................... 57
Figure 5—Text Example I .................................. 60
Figure 6—Text Example II .................................. 61
Figure 7—Text Example III .................................. 62
Figure 8—Unnamed Macro Example ....................... 76
Figure 9—ON Conditions Table ......................... 109
Figure 10—Leaf and Node Information ................... 113
Figure 11—Data Structure .................................. 117
Figure 12—Motion and Time Tables ....................... 124
Figure 13—Pathmov Example I ........................... 127
Figure 14—Pathmov Example II ........................... 128
Figure 15—Names Table ..................................... 131
Figure 16—"FILL" Example ................................. 148
Figure 17—Color Wheel ..................................... 152
Figure 18—Hidden Surface Example ..................... 156
Figure 19—"Origami" Fish ................................... 158
Introduction

Much has been learned about higher-level programming languages in the eighteen years since FORTRAN has been with us. Both theory and usage have shown that to make any implementable computer system adaptable to users and their tasks, a considerable effort must be made in the planning stages to provide for natural user interface (habitability) and to include methods for easily increasing the power of the language in terms of itself (extensibility). These design features are particularly important in a general purpose animation graphics system for two reasons: first, the users of a graphics system are typically not programmers or even scientists; and second, it is extremely difficult to foresee the future requirements of computer graphics.

In discussing the concepts of habitability and extensibility in a graphics system, it is worthwhile to note that most present day systems are not very good examples. FORTRAN is neither particularly habitable nor extensible—yet most graphics applications use assembler routines added to FORTRAN. The majority of the remaining systems use assembler routines stand-alone in an interpretive mode to draw lines, points, arcs, and to display text. The general simplicity of the stand-alone approaches preclude extensibility although most have made some attempts to be more habitable, with good mnemonics for
the commands, for example. However, since any conceivable graphics system must either use a host language or have its own language, the advantages and disadvantages of both approaches must be considered.

FORTRAN is the most common graphics host language for several reasons. It is relatively easy to add assembler routines to FORTRAN to do graphics. Any user input/output is simply and conventionally handled. Numerical calculations of significant complexity may be readily coded in FORTRAN. Most programmers likely to be in the graphics business know FORTRAN and it is the closest thing to a machine independent language in that FORTRAN is most widely available on large and small computers.

FORTRAN, however, is not an ideal language for complex data structure manipulation, parsing language input, or managing critical timing information. A basic difficulty in designing a graphics language is that of retaining FORTRAN's comparatively excellent numerical capabilities (which are compiled) in an interpretive, real-time, on-line system.

One approach might be using PL/1 as a host language for the systems programming and numerical calculation and SNOBCL4 for the data structuring and command parsing. A very large system may be obtained using such a combination. None of these state-of-the-art languages is available on a mini-computer, however.
Naturally, the display part of the system changes at least as often as the computer part so the machine independence and portability advantages gained in FORTRAN may fade when compared to the restyling of the system to allow efficient use of a new "scope." It seems to the author that, given present tools, only a very numerically oriented graphics system for scientists, or a relatively small system may justify using the FORTRAN-based approach. Computer animators, whose numerical needs may be otherwise dealt with, just cannot live with FORTRAN on a mini-computer, mainly for reasons of timing and memory restrictions.

The other approach in common use is creating interpretive line and circle drawing programs which are generally single purpose or limited purpose. Usually, a good idea of the exact application precedes the development, and the program eventually does its job of laying out printed circuit boards or designing tire treads.

Computer animation, as it turns out, is a much more complex problem. The artists—who are as much a part of the animation system as the programmers and the hardware—tend to make demands not readily anticipated by experienced computer people. To expect an animation system to include all features desirable in the present and future as resident functions is as unrealistic as trying to solve all scientific problems with FORTRAN library routines. What is
needed, then, is a language that can be built upon, added to in various ways, and constantly expanded as more people interact with it—an extensible computer graphics language.

However, a language of this type must also encourage interaction by the people who will demand the most from it—the artists, filmmakers and instructors. Every effort must be made to avoid rigid syntax, to use "natural language" input and to provide useful feedback. Reference to complex structures should be as simple as possible, and most of the systems programming aspects should be transparent to the user. Furthermore, since flicker is generally unacceptable (and usually unhealthy), interrupt-level structures must be designed to adjust time and execution priorities in order to maintain a real-time display wherever possible. Storage allocation and reclamation are also critical. In short, a good graphics system must take on most of the problems of a timesharing/swapping system for non-programmers while trying to accommodate both persistence of vision and movie cameras.

The remainder of the discussion here will explore in detail one approach to the problem: the design and implementation of an entire graphics programming language on a mini-computer, with provisions for as much extensibility and habitability as seemed practical. Features of the "ideal" system which turned out to be impractical to
implement in the present system were natural language input, extensibility through addition of unanticipated data types, usability on computers of different architecture, as well as a host of more difficult problems such as accepting spoken word input and synthesizing speech output (to replace the teletype for user communication), and adapting the system to any display system, plotter or microfilm recorder.

Much of what has been learned in the eighteen years since FORTRAN arrived can be thought of as the general "desiderata" of programming languages. Many of these new concepts are relevant to a graphics system which has, in addition, several of its own requirements. Next discussed will be the design features which were considered desirable in an animation-oriented system.

Let's start with the simpler elements:

(a) Basic system functions: In FORTRAN, the basic functions (primitives) include: +, -, *, **, SIN, CALL, DO, etc. In a graphics language, they include something like: DRAW, MOVE, ROTATE, SCALE, DELETE, WARP, etc.

(b) Basic variables: In FORTRAN, these are generally numeric, sometimes alphanumeric. A graphics language needs some alphanumeric and numeric variables, but most of the basic variables are pictures.

(c) Iteration loops: Computers were designed to save program space by iterating. FORTRAN has straightforward looping
mechanisms (ALGOL and PL/I have more sophistication here). A graphics language should be able to support loops too.
(d) Good readability: This helps in documentation. To the uninitiated, LISP is unreadable, FORTRAN is better, and COBOL is better yet. Someone other than the programmer should be able to read the graphics program.

Now to some more esoteric features:
(e) Facility for adding assembler subroutines: FORTRAN has this capability but most interpretive languages cannot swap code at run time. This is necessary for future, unexpected routines and for efficient core usage when these routines can be swapped onto disk.
(f) Disk swapping of code and data: Swapping can sometimes be done in FORTRAN with difficulty. Pictures, assembler modules and graphics language statements should all be disk storable and recallable at swapping speed.
(g) Dynamic storage allocation and reclamation: Often, even if a language allows expansion of core requirements at run time, the operating system does not. FORTRAN is designed to allocate the maximum amount of core any program step will use—an approach which creates difficulties in graphics unless core is very extensive. Reclamation of storage is necessary to keep from constantly running out of core and to maintain some space for swapping.
(h) User-oriented error messages: FORTRAN diagnostcs require a manual to decipher. The habitability of any
programming language depends largely on the quality of its diagnostics. A graphics system for non-programmers certainly needs more than "??35" or "FUSHDOWN STACK OVERFLOW" to help its users.

(i) Good interactive response time: This is more a systems consideration than a language design criterion, though it does aid the user in learning the language. Minimization of response time provides a reason for coding in assembler. Anything that takes more time than a carriage return and is not specifically designed to be slow, is probably undesirable.

(j) A macro facility: This provides for generating new code from old code easily and efficiently. No such thing exists in FORTRAN, and it is primitive in PL/I. APL has this power and every language should have it.

(k) Natural language input: COBOL was an attempt in this direction. Use of "real" natural language falls afool of a number of unresolved linguistic problems. The language input does not have to be "unnatural" though, and this is an important consideration for habitability.

(l) Substring matching: This is difficult and unnatural to do in FORTRAN; PL/I and SNOECL allow it. This facility allows the user to process teletype input in his own way.

Finally, the features especially for graphics which make visual data more programmable:

(m) An arbitrarily complex tree-like data structure: This
allows dealing with groups of sub-pictures and groups of groups of sub-pictures in a hierarchy. It should feed the display in the order specified by the user, or default to a simple linear linked list when not needed.

(n) A facility for naming pictures and functions with alphanumerical names and storing them on the disk with these names.

(o) User-specified sensitivity to analog controls in a natural way: The user must be able to affect pictures by his interaction with analog devices. This becomes a non-trivial problem when one's installation has a three-dimensional data tablet, ten dials, and a joystick, as well as the binary function switches, cpu switches and lightpen to watch (the exact configuration is given in Appendix B).

(p) On and If conditions for discrete phenomena such as function switch states, user-defined variable and dial tests, and teletype substring matches: These operators allow the user to control execution of portions of code synchronously or asynchronously according to his choice of binary input devices, programmable counters, variables, dials, and, with keyboard substring matching, a vast number of possible position independent alphanumerical input strings.

(q) Writing notes to the user when creating segments for interaction with less knowledgeable users: When these segments run off the disk, the user can be carefully stepped through the program if necessary. A fast alphanumerical
display terminal is necessary to avoid teletype fatigue here.

(r) Retention of program steps for re-display and debugging: This is necessary for reproducing sequences for filming and testing.

(s) Accurate and flexible timing capability for specifying rate of motion, rotation, function calling, picture duration, etc.: Timing should be specified as a count or as the result of evaluating a function, or as the y-value of the graph in which x is the time count. This feature is critical for filming and precision animation.

(t) In addition, the language ought to be able to accept any well-coded hidden line, shading, linear interpolation, picture editing, compound rotation algorithm or any other display manipulating technique as if the system designer had planned for it.

With the exception of the natural language input, all the above features were designed into the Graphics Symbiosis System. All are implemented and working, as well as many additional features which are discussed later. At this point, films have been made and several different types of users accommodated by this system. Chapter V has some comments regarding these films and the impact of this work on computer graphics and society.
W. C. Watt, in his paper on habitability, defines a habitable language (with respect to natural language question answering systems) as one "in which its users can express themselves without straying over the language's boundaries" [1, p. 338]. In computer graphics, we deal with the input, processing and output of essentially two- and three-dimensional data. The input is only partially in written language, the remainder being from the electro-sonic-mechanical-analog devices used to create or modify pictures. Therefore, the liberty has been taken here to expand the scope of Watt's "habitability" to include most of what affects "natural" man-machine interfacing in an animation environment.

The quality of being "natural" is obviously relative. Using a paint brush or a chisel and hammer is not particularly more "natural" than drawing with a lightpen. However, inputting all points by keypunching cards is far removed from drawing and therefore "unnatural."

When creating a language for use by non-programmers, then, one does not use System/360 Job Control Language as a model. Neither should one expect a mini-computer based language to derive "DRAW CAR" from "Listen, machine, I could really get into making a car on that tube." Habitability is not thought of as an all-or-none quality, though, so an effort may be made to make the language as usable as
possible by any intelligent person likely to use the system. This naturally leads to a system in which the programmer is no longer required at every step of the way to help the animator or instructor create sequences.

Now let us examine which of the previous desirable features lead to a more habitable graphics system. We will start off simply again and progress from there.

Clearly, if a language is easily readable, with comments and a minimum of strange symbols and syntax, it is probably easily written too, though the converse may not hold, as in APL. To extend this by allowing the user to send himself (or others) notes on a terminal at run time while stepping through a long disk-resident program segment conceptually aids the interaction between man and "tube." Add to this user-oriented error messages with suggestions to alleviate the particular problem, and maybe include some fix-ups for commonly encountered errors, and the machine will be a lot less frustrating to use.

Also not overly complex, but dependent on the data structure implemented, is the ability to group sub-pictures and groups of sub-pictures. This allows reference to a composite picture by a single name and thereby permits the user to translate, scale, rotate, and perform other functions to the composite as a whole, and at the same time. The references to single elements and subgroups of the
composite picture are also retained.

Although less trivial than group manipulation from a systems point of view, permitting compound applications of rotation, motion, and scaling to groups of sub-pictures is a natural capability that might be expected of a graphics system. Compound applications of motion and rotation, for example, would be necessary to move a truck along with a dragonfly and two hornets flying elliptical paths inside the cab. Compound rotation allows helicopters to be flown in figure eights with moving rotors going at independent speeds. Implementing this compound capability is simplified greatly by making the display traverse the data structure in the same way that the user actually creates and interacts with it.

A system is made more habitable if storage allocation and garbage collection operate automatically, simply because the user does not have to worry about it. Certain techniques can be used to compact recollected space and to automatically delete or store infrequently used modules of code or picture data, and automatically recall them when needed again, allowing much more efficient use of core without the user's being explicitly aware of the techniques.

To make the habitat more natural for the user, considerable effort must be directed at letting him specify the interaction between visual or program elements and the
twelve-bit analog devices. These devices can be used to specify variation in any dimension, including time. Furthermore, the system should be capable of storing the values of these devices when asked to do so. Then, on a rerun, the user may sit back and watch the program sequence as it was, without touching anything, or he may again interact with some or all of the devices by pushing a button. The new interactions, as mixed with the old, become a new file which can be replayed in turn.

Similarly, sensitivity to binary devices such as function switches must be natural. For this reason, "ON" and "IF" conditions are included in the language. This feature allows the user to key execution of modules, however complex, to the settings of the function or central processor console switches. "ON" conditions set up pending conditions which execute whenever the condition is satisfied. "IF" conditions are used to poll switches and devices within looped code. Since the switches may be programmatically as well as manually turned off and on, this interaction may become quite complex if the user wishes it to be.

The aim of allowing the user more flexibility than he might ever want is furthered by including a number of arithmetic variables. The user can use these as counters, and add, subtract, multiply and divide them. He can use
them as x,y,z-coordinate modifiers in place of twelve-bit analog devices when so specified. Since testing is a binary operation, he can test the variables against constants or each other to control program flow by using the "ON" and "IF" conditions combined with the usual arithmetic comparison symbols: EQ, NE, LT, GT, LE, and GE.

These variables and the substring matching facility to be described were incorporated originally as an attempt to accommodate the exploration of methods to study, create and distort the human face. It was necessary to be able to accept teletype input like "make the nose bigger" and translate it into something like "SCALE NOSE, A...A=A+30CC." The result was designing a keyboard substring matching module to execute code on the basis of the binary success or fail of a match. With enough patience, a rudimentary second-level interpreter could be developed along the lines of writing SNOECL in PL/I, instead of in assembler. An example (the syntax details are not important here, but they are correct):

```
A=0
SCALE NOSE,A
ON KB='NOSE', DO NCSEBIG
ON KB='EYE', DO EYEBIG

NCSEBIG:< IF KB='BIGGER', IF KB='LITTLE', SKIP 4
  IF KB='LIGGER', IF KB='LGT', SKIP 5
  IF KB='EIGGER', A=A+30CC
  EXIT
  A=A+1500
  EXIT
  A=A+5000
```
EXIT>

EYEBIC:< ...etc....>

If the code above were stored as a function named, say FACE1, the command "ON FS1, DO FACE1" would cause FACE1 to be executed once the first function switch were tapped. As every line is sent from the keyboard, the code checks for the above keywords, and does the processing corresponding to the match, if any. If no match, the program will simply continue until a match is seen. This code will recognize the following sentences:

1) Make the nose a little bigger
2) Make the nose a lot bigger
3) make the nose bigger

and, incidentally,

4) Colorless green noses sleep bigger

which can be taken as you wish.

A further refinement is to let a word position be enforced. For example, "ON KE3='NOSE',..." has the match only succeed on the third word in the line. Less rudimentary linguistic approaches could conceivably be added at a later date, if desired.

Note that substring matching of this type allows users to deal with special purpose applications that the system would otherwise need an enormous dictionary or extra-sensory perception to process. A program dealing with facial features can economically and realistically parse a string
like "widen the mouth" whereas a single general purpose system could most likely not, since it could never anticipate all the possible input strings.

As it turns out, this substring matching facility also leads to a degree of extensibility. Extensibility, itself, is normally an attribute associated with compiled languages like PL/I. This distinction may exist because the difference between macros and subroutines in interpretive languages often becomes nebulous. Nevertheless, if a language has "a method for assigning meaning to new syntax constructions and these meanings are, of course, always in terms of 'old' meanings," [2, p.4], the language can be considered extensible.

With the exception of PL/I and MAD, widely-used compiled languages have not had anything like an assembler macro definition facility. For an interpretive language in which the code is essentially kept in alphanumerics, the addition of macros is done by adding some SWCEOL-like string manipulation and parsing operations. TECO, on the PDP-10, for example, is an editor with macros of sufficient power to generate other editors.

More ambitious extensibility as found in ALCOL 68 [22] or Galler and Perlis' ALCOL D [3, Chapter 4] is not being considered at this time since most of these extensions to ALCOL 60 are designed for the sophisticated and/or systems
Before continuing with a more detailed exposition, it will be helpful to examine some previous animation graphics languages and systems. In this manner, the Graphics Symbiosis System may be later viewed by the reader in the context of past efforts. It should be noted that none of the following systems is comparable to the Graphics Symbiosis System in power or flexibility. Each of these systems advanced the state of the art, however, and the respective contributions should be regarded in that light. Hardware has changed significantly since 1968, a fact that should also be realized in evaluating these systems.

The systems briefly described below will generally be restricted to interactive animation systems, thus precluding microfilm-oriented systems, circuit board and other design layout systems, management information, and any graphics the consoles of the CYBER 70 and S/360 series of computers have to offer. No system prior to 1968 will be considered because the hardware and software did not really start to be significant for animation graphics until about five years ago.

All but two of the systems to be discussed have one deficiency in common—there have been too few films made and
circulated. One would think that a system which took considerable effort and backing to realize would be used for producing film. If a system has been around for a few years and only has demonstration film clips to show, the reason surely is not the worldly lack of applications for computer graphics. The exceptions, of course, are the efforts of The National Research Council of Canada, and the work of the Computer Image Corporation (discussed below).

Systems which rely on random effects to generate abstract films will also not be discussed. It now appears that the age of abstract computer art is past, and that the true use of computer graphics is in portraying recognizable images.

The systems to be explored, then, are those that have attacked the computer animation problem and have had some degree of success.

CAFE [4] is a line-drawing animation system implemented in SNOECL and FORTRAN on an IBM 360/67 and 2250 configuration. The system has an easily learned language which allows simple single-axis rotations, some data structuring, and development of "scenes" or reproducible picture sequences. It also has some good camera commands.

This system has no tactile devices and its error messages are largely limited to "EH?." All parameters must
be numerically specified. The language is not extensible and no macro facility exists. Nevertheless, the CAFE system was an attempt to move away from single-purpose graphics programs.

The Computer Image Corporation system [5] uses primarily analog devices to control video tape equipment integrated into a hybrid computer configuration. Its output is roughly the equivalent of hand-drawn "Hanna-Barbera" work (e.g. the "Flintstones"), and, therefore, represents the state of the art in non-digital computer animation. The system's deficiencies seem to center around the cardboard quality of the two-dimensional images (output of the CAESAR system [6]) and the use of scan alteration techniques which give the Foldes' film "Narcissus" [7] some of its quality and limitations. The Computer Image Corporation has successfully mastered the video tape technology. Another technique to their credit is the use of the anthropometric harness to generate figure motion sequences [5, p.105].

In abandoning the digital approach to computer graphics, they have limited future capabilities severely. The system does not have a programming language, much less an extensible one. Adding new analog hardware to create a new function is an expensive, time-consuming approach. One would hope that the concepts and video hardware developed by Computer Image could be merged with a powerful, extensible,
yet habitable digital system, thus resulting in an environment that could produce large volumes of high quality film, in short periods of time.

Two systems make extensive use of linear interpolation to produce images. First, KARMA [8], which was essentially more of an experiment than a system, produced intermediate frames from key frames using an IBM 1130/2250 configuration. The operations were simple and easy to learn in that only linear interpolation commands were developed; no shading, rotation, scaling, or three dimensional effects were provided. Concurrently developed, and much more successful from the animators' point of view, was the system created by N. Lurtnyk and H. Wein of the National Research Council in Ottawa [5]. Conventional animation techniques were used as a basis for an interactive system exploiting shadable key frame animation algorithms. The system is oriented towards professional animators and, presumably, would not be particularly habitable in an educational environment since the commands often use keywords and concepts that are familiar only to animators. The language used is also not extensible, and the tactile devices are limited to a 2-D data tablet.

Output from this system has been impressive. The hidden line problem has been simplified by establishing layers of picture motion so that the processing necessary to
recent work [10] shows the use of linear interpolation to approximate walking and other human motions in a realistic manner. One would like to see the N.R.C. members acquire more sophisticated hardware to carry on their work.

The interactive movie making system developed at the Moore School of Electrical Engineering [11] is an example of an interactive system written in FORTRAN, with a fairly primitive scope (DEC 338) and slow computing equipment (Spectra 70/46). The system is not particularly habitable due to the lack of analog input devices and the FORTRAN implementation, and is not extensible. It is a system usable by scientists for producing science films by coding pictures and specifying time and motion sequences algorithmically in FORTRAN. This system is not suited for non-scientists unless aided by a programmer, and is thus an example of how FORTRAN can severely limit the use of an animation package. In what it was intended to do, however, this system could be called a success.

Another FORTRAN-based approach is Anderson's work in CAMP and its extensions CAMPER, HICAMP, and HICAMPER [12]. CAMP and the later versions have a novel list processing concept for storing pictures in push down stacks. The list processing technique was used to allow pictures of any length to be stored without wasting core, the analogy being
to the variable length number fields in earlier computers (IBM 1401, 1620, etc.). Pictures can be grouped to one level using the "stacks" and the stacks can themselves be manipulated like pictures. This approach made it possible to allocate picture space and create a simple hierarchy within the constraints of FORTRAN. In general, the list processing and storage allocation techniques here are considerably less sophisticated than found in the Graphics Symbiosis System. Limitations of this work are that motion dynamics must be expressed by equations and input must initially be punched on cards. The system is not extensible either. Several good features exist, however, and these included the capability to generate 3-D perspective drawings in stereo and the concept of the data structure used [13].

GENESYS is a system developed by Ron Baecker at M.I.T. [14] which represents a very habitable computer animation package. The user communicates with the system by using a data tablet to draw, specify motion and timing curves and select commands from a menu. Motion and timing sequences can have variations in them which the user can easily specify by replacing one picture or motion sequence for another in a given time slot. However, the methods used to create animation are fairly primitive and cannot really be extended to larger systems with better hardware. The chief contribution of GENESYS is the high level of user interaction provided.
The picture quality of GENESYS leaves much to be desired—all images are composed of large dots, cannot be shaded or represented in three-dimensions. The system has no real programmable language and is not extensible in any way. GENESYS does represent the first computer animation system for naive users, and, therefore, is a valuable contribution to the world of graphics.

This introduction will be ended with a statement from Ron Baecker, a plea which is now hopefully answered to a great extent by the implementation of the Graphics Symbiosis System:

What is needed, I believe, are some fresh ideas, a thorough rethinking and reformulation of fundamental concepts for describing images and animator actions, and perhaps a touch of formalization. The concepts of conversational languages, extensible languages, and picture grammars provide some new directions that I think will be of assistance. [15, p.283]
Chapter I—Habitability in a Graphics System

In the Introduction, several methods were discussed which would help provide a certain degree of natural interaction within a computer graphics animation environment. The purpose of this chapter is to examine in detail the actual features and capabilities of the Graphics Symbiosis System which lead to reasonably natural interaction.

In an attempt to define "habitable," one must remember that "habitable" can be used as an absolute. A habitable system, in this usage, would be one that only required the user to think of his problem, conceptualize his pictures, imagine the paths, and the system would produce film in colors and perhaps stereovision, without so much as the user's physical presence, his mental "presence" being sufficient. There are no truly habitable systems, obviously. A reasonably habitable system, on the other hand, is one whose conventions are easy to learn and whose commands are powerful, yet easy to use. A truly uninhabitable system would be one that required the user to program all his requests in binary machine code, and a fairly uninhabitable system might necessitate describing all interactions within FORTRAN syntax.

In a graphics system, habitability is best realized
qualitatively by providing a high degree of interaction with the images and equipment. To attain a high degree of interaction, two design criteria must be implemented. First, all available methods of communicating with the electronics must be exploited. Tactile devices are particularly important since they provide direct communication. Where these are insufficient, language may be used, but it should be easy to learn and use. Second, the results of the interaction should be immediately visible—essentially the idea expressed by the concept "real-time:" "that is, time which is 'real' because the moment of the artistic idea is also the moment of its materialization. This is a revolutionary idea in film-making, rather like editing the film before it is made. It makes the creative process a very spontaneous thing, for all the problems of execution are solved in advance" [32]. A "real-time image" is one whose display is complete in less than the one-fifteenth of a second that determines the threshold of persistence of vision. Accordingly, system execution priorities must be adjusted to maintain a flicker-free image or the interaction gained by clever language and device usage is lost.

One must also decide how free the language will be, that is, what syntax requirements will be imposed upon user input strings. If the power of a language is defined as the range of programming tasks that can be attempted within the
syntax, one can note that a rigid, unhabitable language may be just as powerful, more powerful, or less powerful than a language that accepts a subset of English sentences. There is no real correlation between the power of a language and its habitability (although execution speed may be sacrificed by excess syntax processing). That few people can use a language does not affect its power, only its use. If one wishes to maintain a real-time system with reasonably instantaneous response time, it is necessary to require some sort of easily parsed syntax that does not need minutes of directory searching and linguistic analysis. It seems, at this point, that a language with easily remembered keywords and predictable syntax is the best approach.

Rotation, for example, is a complex manipulation when compared to scaling or translation. It requires calculation of sines, matrix multiplications, and some housekeeping. An unhabitable system would require the user to write a matrix multiply routine and a sine calculation routine and apply it to the data point-by-point. A FORTRAN-based system might supply the matrix multiply and sine routines as subroutines, or even a subroutine handling rotation within FORTRAN "CALL" syntax, thus only requiring a knowledge of FORTRAN. A reasonably habitable system should recognize "ROTATE" as a command and use its alphanumerical operators to set up the sines, matrices, and do all the housekeeping. A habitable system with tactile interactions also allows the user to
specify the operands as tactile devices, or as numbers, if so desired. Finally, a system with good error messages tells the user if he has incorrectly specified the axis or speed of rotation, for example, and thus helps him to learn the syntax and avoid further errors. The term "habitability" contains the root "habit," and a habitable system should be one that leads the user in forming (good) habits as a result of his interactions. A more habitable system simply achieves this goal with less pain on the user's part.

Any interactive computer graphics system that is even barely usable has at least one feature that makes it more habitable for non-scientists (in particular) than other computer packages—the visual feedback. This revelation is both startlingly obvious and profound when thought about. For instance, the visual feedback in an interactive graphics system instantly tells the user if things are working correctly, in at least ninety-nine per cent of the pictures and picture sequences he will attempt. The user is capable of pattern recognition (to say nothing of aesthetic judgements) far more sophisticated that the machine generating the image, and he is thereby much more in the position of knowing whether the picture is "right" or not.

To explore this point further, consider the possibility of a simulated computer graphics programming language,
written on a high-speed super-computer. The purpose of the simulation would be to allow preparation of code that would produce animation sequences, at the low cost of hooking up a terminal to the super-computer. No scope would be permitted, however, since interfacing is so costly on big machines. Later, the sequences would be filmed on the mini-computer system being simulated. What could be done with such a capability? Unfortunately, very little; the visual feedback is so important to animators, and coordinates listed on a teletype are so nearly meaningless to anyone but the system programmers, that it would be very difficult to do more than syntax checking of the language commands. Any kind of interactive data generation capability would be lost; rotation effects are not perceivable from matrix values; and even the aesthetics of size relationships is left purely to the imagination.

The point is that although the display and the computer are number-oriented, many people likely to use graphics are not, and that computer graphics without more or less instantaneous visual feedback necessitates accepting a far less productive environment and one that is much less attractive to the animator than a system like the Graphics Symbiosis System. Computers have the image of being both frustrating and remote—an image which can be overcome only by making interaction less difficult and more tactile.
Most interactive graphics systems have a further inherent capability which helps make them habitable—the interaction itself. The tactile devices one uses to affect the image on the screen take many forms—teletypes, dials, joysticks, data tablets, function switches, etc. Let us discuss how the Graphics Symbiosis System handles these devices. (See Figures 1 and 2.)
Figure 1—Environment

ENVIRONMENT. A) Working with the sonic pen—a 3-D data tablet. B) Controlling an image with the light pen and dials.
Figure 2—Devices

Figure 2—Devices (cont.)
One may conceptually separate the above gadgets into two categories—binary devices and analog devices. The former carry on/off values and are useful for controlling discrete phenomena. Of course, if no other way is available, a function switch may be used to affect small changes at the expense of much button pushing (or many buttons). Function switches are ideal for controlling execution of animation sequences, or for initiating standard housekeeping tasks, for example. Analog devices are somewhat rarer on display systems, at least in quantity. Typically, such a device will deliver nine to twelve bits of information, which, when scaled to the size of a sixteen inch by sixteen inch screen, comes out to an accuracy of $1/32$ to $1/256$ of an inch. The Graphics Symbiosis System makes use of some sixteen analog and thirty-three binary devices. There are ten dials which are referred to as "$D0$ through "$D9,$" a joystick whose axes are called "$JX,$" "$JY,$" and "$JZ,$" and the three-dimensional data tablet "$SP$" (for sonic pen) whose axes are known as "$SX,$" "$SY,$" and "$SZ.$" As a group, with the variables to be discussed later, they are all called "DEV's." A "DEV" is, therefore, a twelve-bit analog device, and anywhere in the syntax that a "DEV" is specified, the user can choose any of these devices or variables. Thus the user can specify which dials control the rotation, path following speed, scaling, zooming, translation, intensity or zoom point of an object, to name a
few capabilities. He can also use the sonic pen or joystick axes to transmit these values. DEV’s may be multiply assigned, thus giving an easily defined correspondence between x-displacement and size, or intensity and rotation speed and y-dimensional scaling, for example.

If the analog input devices were unavailable, a very perceptible amount of the system’s habitability would vaporize. At least nine bits of information must be supplied somehow and the analog devices are the best suited from a man-machine interaction point of view. Function switches may be repeatedly tapped or set in groups of nine or twelve for each function, but interaction is somewhat less rapid and much less natural. Four digits or two-letter codes typed on a teletype provide enough bits but every change requires re-typing, an action which is only marginally tactile and quite removed from the "feel" of turning a dial or moving the sonic pen. In other words, habitability, being derived mainly from natural interactions, requires interactions to be as natural as possible, within restraints. If the programmatic or financial restraints do not eliminate analog devices, the user benefits greatly.

Since the analog devices are read left-justified into memory, the accuracy lost is on the low-order end, thus allowing fairly accurate comparisons between the nine-bit
dials, the eleven-bit sonic pen and the sixteen-bit variables ("A" through "Z"). By using variables set to dials, for example, the user can easily make other than one-to-one correspondences by looping arithmetic assignment statements. Let WITCH and FISH be picture names:

```
SCALE WITCH,DO
SCALE FISH,A
A=DO/2
```
every time this code is executed, it will set the witch to the fractional size indicated by "DO" and the fish to half that size.

The above example serves to illustrate one further advantage of the system: certain commands, namely those whose dynamic control is important, are only required to be set up once. Thus, scaling, translation, arbitrary-axis three-dimensional rotation, intensity, cutoff plane setting and zoom point location are commands which assign the device or devices to the name mentioned in the command. "SCALE WITCH,DO" will constantly cause the witch to be scaled on "DO" even though the command is executed only once. "SCALE FISH,A" also allows any change to variable A in the future to be reflected as the size of the fish. To maintain a correspondence between the variable "A" and the device "DO as in the three-line example above, only the line "A=DO" must be re-executed. The right side of an assignment statement may have any combination of DEV's, decimal
numbers, function switches (used as Boolean values) as long as they are separated by the normal arithmetic operators (+, -, *, and /). One could have variable A represented as the result of a system of equations whose independent variables were dials even though this system is not particularly well-suited to numerical analysis, since it does not compile code. Note that the "FIX" command will remove DEV assignments and freeze the picture at that point, whereas the "RESET" command will reset the scale, translation, rotation, intensity, etc. to the original positions.

If each command had to be continuously set up, operation of the system in line-by-line interpretive mode (asterisk or non-macro mode), would be impossible, and real-time motion without macros could not be achieved. Much of the system's habitability is derived from the ease with which users can set up constantly changing rotations, scaling, translations and so on, with single commands issued only once. The naive user will spend most of his time in non-macro mode and most film is generated in this mode at this point, so the system must be, and is, easiest to learn and use at this level of usage.

Variables are also useful as the image complexity mounts, causing one to run out of dials—something quite easy to do, considering it takes five dials to do an
arbitrary axis rotation, three for translation, three for single axis scaling on all three axes, three for zoom point, two for cutoff plane, and one for intensity setting. The user can turn the dial he chooses until the desired effect is obtained and then set a variable to the dial, retype the command with the variable as the Dev, and the dial is automatically deallocated.

Variables may also be used as counters for timing control, as coordinates to generate new pictures or algorithmically modify old ones. Variables can be assigned values to alter program control, although it is just as simple to use function switches which can be programmed to turn off and on.

Without variables, two problems would immediately appear. The first is that without variables, users would be absolutely limited to the analog devices. Once these were exhausted, the user would have to limit future interactions severely. Variables can substitute as placeholders for analog devices and therefore allow the user to manipulate many objects in a complex image. Second, variables are used in macros where they are necessary for passing results, algorithmically creating and modifying data, and setting up conditions for which the user need not be called upon to adjust dials. It is hard to conceive of a programming language without numeric variables. Treating programmed
variables as analog devices and allowing both analog and
digital (algorithmic) settings seems to be a valuable
concept and is one that helps the user get himself out of
difficulties and explore new techniques through macros as
well.

The other tactile device which the system uses is the
display terminal, called the V105 by the manufacturer (see
Figure 2). Users enter commands, macros and values; they
are prompted by this device, receive error messages on it
and can take corrective action on it. In other words, it is
the teletype assigned to the scope user. If a teletype were
not available, inputting statements interactively would be
difficult. An approach that could be used is the "menu"
technique whereby the user points with a lightpen at
keywords appearing on the scope, and the system decides
which command was chosen on the basis of the lightpen hit.
This method is difficult to use if the user has to input any
alphanumeric strings, such as new names. Of course, a scope
could have a picture of a typewriter on it and the user
could hunt-and-peck "type" with the lightpen but this would
be fairly time consuming. Many systems have keyboards
available which are not really teletypes because they go
directly to the scope hardware rather than through the host
computer, with the characters echoing on the scope's screen
through a hardware character generator. Keyboards are
normally just as useful as teletypes, except that for
animation purposes, it is nice to have the whole screen to film without portions dedicated to listing commands and options. With a system as rich as the Graphics Symbiosis System, there might be very little picture space left.

Text input as commands can be entered in relatively free form with respect to spaces, tabs, and commas. The operators and operands of a command are definitely position dependent, but some care has been taken to make the ordering natural, or at least predictable. Comments may be interspersed by the "REMARK" command or by use of a semi-colon before the comment.

Several different levels of teletype interaction are provided the user, and a key character is echoed to indicate each mode of input. Normal mode— that is, system command level—is indicated by an asterisk ("*") at the beginning of each line, and will be referred to as "asterisk mode." The key character, by the way, also indicates readiness to accept input. Asterisk mode is the default mode to which the user starts and remains until he types a command which triggers another mode. It is also the level to which the panic button, a "^C" on the teletype, and the "EXIT" command trap.

If, during a creation command ("GETDSK" primarily, but also "SHADE," "PUTPOI," and "FILL"), the user pushes function switch fifteen (FS15), the creation command will
interrupt and allow the user to type commands to alter the partially completed picture. Typically, he might rotate the image, or scale it, for example. When he types "RESUME," the creation command continues until finished, unless of course he pushes FS15 again. This mode is indicated by a pound sign ('#').

The modes of input were developed as time progressed and needs appeared. It became clear that the key characters help the user to recognize at a glance the mode in which he is operating. Without these characters at the beginning of each line, one would never be sure of the operational status of the system. These mode indicators are simply a type of feedback that lets the user know whether he and the system are in agreement, something he should always know. Providing the user with feedback of this type adds to the system's habitability, although perhaps in a minor way. It is surprising how many "minor" features such as this one were included during development to make things easier to learn and easier to use, by providing meaningful feedback.

Often during the execution of a macro, it may be necessary to get teletype input of one or several lines, or the name of a picture or value of a variable. The macro author has the option of using either the "WAIT" command to put him in period ("." ) mode, or the command "INPUT," which waits only for a single name or value to be typed. Once
"WAIT" is issued, the user must type "RESUME" to have the macro continue, unless an operand is specified as the number of lines for which to wait. Thus "WAIT 3" would allow and require exactly three lines of input before continuing, whereas "WAIT" would allow an indefinite amount of input.

"INPUT $A" (where $A through $H are permissible) requires a name to be input. "INPUT A" requires a numerical expression to be typed in before continuing. By using "INPUT," general purpose utility macros may be written to apply to any name, with the macro querying the user for necessary input as it executes. Several examples of reasonable complexity will be found in Appendix C.

While creating a macro on the display terminal, the user is reminded that he has not yet supplied enough matching right angle brackets by a plus sign ("+") appearing at the beginning of each line, thus indicating macro-creation mode. These plus signs also appear in multi-line commands, of which "TEXT" is a good example. Once the user specifies enough right angle brackets, the system reverts to its previous mode.

Error messages in the Graphics Symbiosis System have been carefully developed to aid the user in learning and interacting with the system. Some redundancy in syntax specifications permits typographic errors to be indicated as such, rather than let them unluckily fall into a strange
syntax mode and produce undesired results. When an error occurs, the user is first notified by "??ERRCR xxx." He may already know what the number means, but before he so much as blinks, the line in error and a following line with an arrow pointing at the error are typed out. If the user understands the error at this point, he can skip the next step and take corrective action. If, however, the error needs explaining, he types "?" and a disk file is searched for the error message corresponding to the number. If the user is not in asterisk mode, as when executing a macro, for example, he is put in question mark mode, indicated by a "?" in which he is able to input a line, or a number of lines of code to replace the one in error. When he types "RESUME," he can reenter the macro at the instruction which follows the one that failed. By just typing "RESUME," he can, in effect, skip the line in error. If in asterisk mode, the line is cancelled and the user remains in asterisk mode.

At this, as well as at any other point, the user has the option of consulting the "HELP" file by using the "HELP" command. In any mode accepting teletype input, "HELP" gives the names of all the system functions. "HELP HELP" explains the system conventions and how to use the help command, and "HELP" followed by a command-name types out the syntax of the command indicated, the switch options if any, an explanation of what the command does, some helpful hints, and at least one example for commands that have operands.
The user may type "O" to abort the typeout if he has seen enough. Pages come out until the terminal display screen is full, at which point the user must hit the carriage return button to advance the typeout to the next half page. The "HELP" file is listed in Appendix A.

During the preparation of a macro, it might be helpful to use error interrupts creatively to input multiple lines of code. This particular facility leads to the name "creative error messages." All the macro author needs do is type a funny character in a line and the error message mode could be entered. Of course, "WAIT" could also be used.

Without error messages of any kind, the system would be usable but not very habitable. During early development, processor halts were used to indicate errors, an approach which soon lead to the idea of ignoring errors. This, too, was a poor idea since one never quite knew where errors were occurring, although the visual feedback made it obvious that something was wrong. Quickly implemented (as part of the original master plan, but with increased priority) was an error reporting technique that printed out the program counter contents of the error location. This helped development, but was predictably recognized to be useless to users who have no way of correlating core locations with error meanings. The present error reporting system was developed and is certainly much nicer to use. "Impossible"
errors, that is, ones occurring only as the result of drastic system failures are reported with the program counter contents and the message "SYSTEM CRASH—CALL THE SYSTEM'S PROGRAMMERS!"

As it stands, the error messages provide good diagnostics. The little arrow under the syntax element in error is very useful, and the disk-resident explanations help the users learn the system. A new step in user interaction was probably taken with the inclusion of "creative" error messages. In any system, diagnostics are very important for establishing habitability, and a system aimed at non-programmers derives even more of its habitability or lack thereof from its error messages.

In concluding the discussion of tactile and keyboard interaction, it is necessary to note the panic facilities of the Graphics Symbiosis System, which can also be used creatively. The function switch interrupt button—marked "panic" on plastic overlays that fit over the buttons—will interrupt any macro at the next statement and return the user to asterisk level. Many long or time-oriented routines like "SMCOTH" or "PATHMOV" also check the panic button in their innermost loops. On the display terminal, a "C" will abort anything as soon as any input or output is requested, and also return the user to asterisk mode. "C" stops output to the terminal display screen and leaves the user in
his current mode. "^U" allows the user to cancel an entire line that he has typed, and the "rubout" key will backspace over the last letter typed, erase it, and replace it with the next character typed. Furthermore, the user could program any device or function switch in the system to cause the "EXIT" command to execute, thereby programmatically simulating a panic button push.

An example of creative use of the panic button is the following macro which aids in adjusting the joystick to zero with the rheostat trims:

```
<PROMPT JX,JY,JZ; THIS PRINTS THE VALUES OF ;JX, JY, AND JZ.
SKIP -1>
```

This macro would go on forever unless the panic button were pushed.

A habitability consideration aimed primarily at naive users is maintaining the impression that they are controlling the system when in fact the system is simply permitting them to input alphanumerics, adjust dials, and push buttons. User panic facilities make the system less scary to use, since even adventurous macros and experiments can be tried and later aborted if necessary by a single button push in most cases. It is often much easier to write a macro without initially specifying exit conditions and rely on the panic button for escape. A system in which the
user has positive control over program execution at all times is easier to learn and use than one which allows no non-destructive escape from accidental user-caused infinite loops.

It should be noted here that the user does not need to know all the commands, control characters and tricks of the system. He only needs to know a subset of the features roughly matched to the complexity of his problem. Most user interaction is at asterisk level and only five or six commands are necessary to satisfy the demands of many naive users. In fact, much good film was made on the system before anything other than asterisk mode existed.

So far, we have dealt with the details of the habitability considerations which concerned interaction with the programming language via keyboard, devices and buttons. The Graphics Symbiosis System allows levels of complexity not only in macros, but in structuring the displayed pictures as well. Chapter III has a detailed description of the mechanics of this interaction.

To a computer scientist, the notion of a tree structure is basic. He might have trouble explaining the utility of trees to an artist or filmmaker, but at least the idea is well understood by him. The notion of a tree can be fairly easily introduced to graphics users once they have used the system a little while. They ask if one can group items for
reference purposes. Soon, they want to control size, translation and maybe intensity of a group of items in some kind of hierarchical scheme. Once they know such grouping can be done, they want to learn how to do it. Visual feedback is also a fine motivational influence. Furthermore, once the notion of compound rotations—propellers or flapping wings on objects that are moving—is introduced, the small burden of doing "housekeeping" for the data structure, given suitable aids, is accepted.

The capability of grouping picture elements to any level and applying any transformation on any or all levels of the structure in real time is easily the single most visually exciting contribution of this system to the world of computer graphics. Half-tone birds have been taught to fly, airplanes flap their wings, and a witch flies around with a propeller providing the pull. Even walking motion, when analysed, is really a series of bounded rotations around joints, nested to four or five levels. Control over each level is, of course, assignable to analog devices or to the variables so the interaction may be tactile, programmatic, or both.

How, then, does a person to whom tree structures mean nothing interact with this powerful tool? He simply is required to use the "GROUP" command with the appropriate
operands. For example, given two pictures as leaves—AIRPLN and PROP— the following sequence would fly the AIRPLN around with the PROP spinning:

```
GETDSK AIRPLN ;GET THE AIRPLN FROH THE DISK
GETDSK PROP ;GET THE PROP TOO
GROUP PROP,AIRPLN,PLANE ;GROUPED STRUCTURE IS
 ;NAMED "PLANE"

ROTATE PROP,X,DO ; PROPELLER SPEED ON DO
ROTATE PLANE,Z,D5,D6,D7 ; ARBITRARY AXIS ROTATION
REMARK /SEE APPENDIX A FOR SYNTAX HELP/
```

A little more (but not much) work is necessary if the position of the propeller is not originally correct, as is the case when the same propeller is attached to the witch's broomstick. All that is needed, however, is simple scaling and translation to adjust the propeller position and size. Figure 3, which follows on the next four pages, is a photo-scenario illustrating the use of grouping and other system functions.
GROUPING -- A) First the user gets the witch outline and shades it, B) Then she gets the propeller, groups the shaded outline (SWITCH) with the PROP and requests a diagram of the data structure with "TREE". 

C) After checking that the data structure is all right, she scales the propeller to the correct size, sets it rotating and moves it into place with the joystick. D) Then the entire group is rotated and a directory of the picture elements is requested.

E) The directory gives the devices assigned to each picture and group. F) The final grouped image can be manipulated as if it were a single vector list.
Figure 3—Scenario (cont.)
Figure 3—Scenario (cont.)
**Core Directory**

<table>
<thead>
<tr>
<th>Picture Size</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>WITCH 688. PX</td>
<td></td>
</tr>
<tr>
<td>SWITCH 2880. PX</td>
<td></td>
</tr>
<tr>
<td>PROP 320. PX; SC-D9; MO-JX, JY, JZ; RD-D8</td>
<td></td>
</tr>
<tr>
<td>SAM 0. PX; GR; RD-D3, D4, D5, D6, D7</td>
<td></td>
</tr>
</tbody>
</table>

5888. Words used in 4 pictures
5888. Words free in 3 fragments

---

**Figure 3—Scenario (cont.)**
The user builds groups into hierarchies by grouping them (see Figures 10 and 11—Chapter III). A command "TREE" lists the structure on the display terminal with tabs to indicate the grouping level of each element. To insert and delete elements from a formed group structure, the "PUTLIB" and "GETLIB" commands are necessary. When the user specifies the first two operands of the group command, he is tacitly including all group nodes or leaves in between the two, if any, although the order in which he specifies them is not important. Thus, if he makes a mistake and groups too much—and this is immediately visually evident if any group transformation is applied—he simply "PUTLIB's" the entry in error. The "PUTLIB" command selectively blanks the image but saves its core image and structure, and chains it to the in-core library data structure. The entry, which can be a group (node) or a single picture (leaf), is also removed from the hierarchy of displayed pictures. The entry could also be deleted, but then its core image would be destroyed. A "GETLIB" command will redisplay the picture, placing the entry in front of the data structure if the leaf or node name is the only operand of the command, or after any element in the entire displayed data structure if that element's name is the second operand. Thus, with "GETLIB/PUTLIB" sequences, the picture may be moved around within the data structure quite easily. One always knows which element he is moving because it is temporarily
blanked. For example, in the above code, "GETLIB WING,PROP" would put the "WING" into the group "PLANE", whereas "GETLIB WING" would display the wing, but not place it in the group, assuming "WING" was put in the in-core library at one point or another. Then, "PUTLIB WING" could remove the "WING" from the displayed structure and blank it, removing group assignments, if any. Note that, for all practical purposes, the order of sub-elements of a group is unimportant unless some of them later have to be subgrouped. A group may also contain just a single leaf, a technique which is useful for creating complex tumbling sequences which require multiple-level rotations.

It must be reiterated that the capability of specifying something as powerful as grouping so easily is an important addition to the system's habitability. Were this capability not developed, the ordinary user simply could not effect compound transformations unless he resorted to some special assembler coding, an unacceptable approach for the non-programmer user. Providing easy access to the power inherent in a tree data structure for even relatively naive users is a design feature that was deemed necessary from the earliest stages of planning.

Details of algorithmic implementation of grouping and compound transformations will be found in Chapter III.
Also of convenience to the user are the disk access and file manipulation commands. "GETDSK" and "PUTDSK" commands are written so well that it is now considerably easier to use the Graphics Symbiosis System to do general purpose file manipulation than to use the program supplied by the computer manufacturer. The user has the choice of recording and retrieving his pictures in binary to save space, in easily readable decimal format, or in Vector General display instruction format. Renaming disk files is as easy as renaming core resident names. A background job was implemented to print any disk file on the system hardcopy device while people are using the system for graphics. Taking advantage of the Disk Operating System's file structuring capabilities, the system also allows access from common areas and requires a password log-in for integrity of the user production disk cartridge. Macros can be fetched from the common area by the "CALL" command, and from the user area by the "DO" command. Disk directories may be obtained for any area of the disk ("DIRDSK" and "DIRCO\text{\textit{om}}") or for all of the disk ("DIRALL"). The "HELP" command in Appendix A further explains the disk oriented commands in the system.

One exciting capability of the system that resulted as one of the many interactions between users and systems programmers was the variable intensity setting of grey-shaded areas. The extremely flexible system shading
routines include two dimensional and three dimensional shading at any angle (or optimized angles), with any raster for spacing, and, by user interaction with the "SETINT/S" command, any grey level. By specifying a dial, the user can vary the intensity of any surface of a shaded object, one surface at a time, until the desired effect is achieved (see Figure 4).
SETINTENSITY  — The graphics language allows the user to interactively specify the intensity of each facet of a 3-D shaded picture. The intensity is normally set by a dial whose variations can run through all 32 levels of intensity.
If the system were coupled to a DATACOLOR colorizer[23], the artist could look at the colorizer's screen and the grey level he chose could be transformed into a rich, saturated color. By varying the dial, a whole range of colors could be investigated.

Other interactions with the system software architecture include a start on data generation and editing facilities ("WARP") and the "FILL" command. "FILL" allows the user to specify an outline, a fill character, and a speed of filling. Filling is done by positioning the character randomly in the outline until a function switch push terminates the fill. The orientation of the fill character may be specified as randomly rotated or parallel. Thus, grass can be grown on a hillside, trees fill a forest, hair grown on a billiard ball, or feathers appear on a bird. The potential of this command should be limited only by one's imagination—a nice feature for a computer package.

Text on the screen is handled unconventionally in the Graphics Symbiosis System. All characters are software subroutines and are the result of conversions of Alan Hershey's [16] beautiful set of some fifteen character fonts. Since these are software characters, they may be rotated, scaled, and translated like any other picture elements—a level of flexibility not to be obtained with hardware characters which bypass the analog hardware (see
the Frontispiece and Figures 5, 6, and 7).
БГЕЗП
АВДЖИ
АВДЖИ
БГЕЗП

carussian

Figure 5—Text Example I
map symbols

Figure 6—Text Example II
stop watching

TV

makes people sick

Figure 7—Text Example III
Habitability, as defined in the Introduction, really appears both actively and passively. So far, the active aspects of the system's habitability—that is those apparent to the user—have been discussed. There remain, however, many features in this system which are invisible to the user in most cases but which actually allow the active features to exist and function smoothly. These could be called the passive habitability considerations.

Probably the most important internal algorithm to the system's functioning is the storage allocation and reclamation routine. Thirty-two word blocks of memory are kept in the largest possible chunks (up to 4,096 words each) without shuffling by the reclamation routine. Allocation is done by a best-fit method if the size of the new element is known. When the size is not known, that is, in the case of dynamically growing allocations such as drawing new pictures or shading or creating new macros, the largest block available is taken. Any leftover is always returned to the stack. Chunks are automatically chained to permit pictures of larger than 4,096 words. More detailed discussion of the allocation and reclamation routine is found in Chapter III.

Other passive habitability features include handling all the interrupt-level activity, directing the data structure traversal by the display, input syntax checking, names table building, device polling and other systems
programming tasks. Most of these algorithms are also more fully discussed in Chapter III.

One could define a hierarchy of users too. A "first-level" user understands the concepts of graphics and can write macros fairly readily. A "second-level" user is a beginner who is comfortable only with simple commands. A "third-level" user knows nothing of the system and his entire verbal interaction is with strings of some natural language (presumably but not necessarily English). Almost anyone who has the dexterity to operate a typewriter in any capacity fits into one of these three categories.

At the risk of leading into the next chapter, one more concept incorporated in the system that makes it easy to use must be mentioned: macros are user-oriented too. For the sophisticated ("first-level") user macros are easy to write, easy to learn to write, easy to follow, and easy to document as well as being powerful. The "PROMPT" and "INPUT" commands make it simple to step a "second-level" user, however naive, through a complex macro. Substring matching of keyboard input will allow macro authors to anticipate natural language responses from "third-level" users, at least in a rudimentary fashion. It is in providing access to a graphics system for these second- and third-level users that the true beauty and habitability of the system is found.
A project for the near future is a long tutorial macro to aid in teaching the system to beginners. It might even exhibit some of the computer assisted instruction capability of the Graphics Symbiosis System.
Chapter II—Extensibility in a Graphics System

In the last chapter, the concept of habitability was broadened so that it could apply to most features that make a system easier to learn and use. Likewise, "extensibility" has been broadened by programming languages theorists. J.A. Feldman commented in a panel on extensibility: "I thought I understood it before I came here and I will first tell you what I think has happened here. It seems to have become 'all useful things to do in helping to write systems programs.'" [17, p. 53]. The software in a complex graphics system has many of the problems of a time-sharing operating system. If this software allows extensions to be written in terms of itself, thus adding new "system" functions as well as new user-oriented functions, it would rightfully earn the descriptor "extensible."

It would be helpful to have some programming language background to make this chapter readable.

Extensibility in interpretive languages is a somewhat nebulous concept. Extensibility usually refers to compiled languages in which compile-time procedures are used to expand prototype macros into code, change data types and add new operators all at the one-time expense of more compilation. The object code will run as if there were no macros, since all the macro processing has already been
done. Interpretive languages, however, never get rid of the syntax checking and macro generating code and all the extensible features must be processed at run time, often at great efficiency losses. For this reason, interpretive extensible languages have not been used for systems programming because no one wants a large and powerful interpreter taking up great amounts of memory and computing time. Other applications of programming could benefit from an extensible language in interpretive form and the following discussion will attempt to clarify why this very approach is acceptable in an interactive graphics environment.

If the basic system functions (often called "primitives") of the language are very simple, say, on the level of adding, bit testing, or shifting, the interpretive overhead for syntax checking is going to be very costly as compared to the machine code eventually executed to do the primitive function. If the primitives are on the level of complexity of, say, matrix multiplying routines, the interpretive effort starts to be effective since the overhead is less out of proportion to the eventual "work" done—it takes more instructions to do the matrix multiply than to do the syntax checking. For example, Comba's unimplemented design of a manipulation language for abstract geometry [34] includes complex basic functions for model building, specifying hierarchies and orienting drawings in
three dimensions. The primitives of the Graphics Symbiosis System are very powerful and each represents a large number of individual calculations. For this reason, interpretive statements and even macros are reasonably efficient, which is fortunate because the interpretive mode of operation is so necessary to interactive applications. These observations are supported by the comments of Stephen A. Schuman in a paper on extensible interpreters:

...an interpretive system is not without its accompanying drawbacks, which are all too often cited as sufficient reasons for abandoning this sort of approach altogether. The most serious arguments, of course, are based on considerations of efficiency. There is inevitably a certain amount of overhead associated with interpretation, though it is not necessarily inordinate. The proportion depends on the disparity in "level" between the abstract machine which is realized by the interpreter and the actual machine on which it is implemented. When the difference is substantial, an interpretive implementation can compare quite favorably with any other conceivable approach. [18, p.120]

Thus, within the Graphics Symbiosis System, the level of the abstract machine—that is, the language for graphics—and PDP-11 machine code is very great and therefore the macro language would appear to be reasonably efficient.

For example, when the drawing routines in the system were replaced by macros (see Appendix C), the macros operated sufficiently fast to be able to trace the sonic pen
or joystick in real-time. Furthermore, the joystick drawing macro only takes six hundred words of storage in ASCII format, yet the original assembler command took much more. Also, it is quite a bit easier to add features to macros than it is to code up new assembler routines or modify old ones.

The system was never designed to do complex mathematical tasks at the macro level, simply because the primitives for arithmetic are so close in operation to actual machine level instructions but need to be interpreted as text at nearly the same cost as "ROTATE" or any other command. The arithmetic processing is not so slow as to be useless, especially for the level of mathematics one expects of non-scientific users, but still represents something less than a good computational facility. In other words, the primitive for addition might take three hundred instructions for the realization of what a single machine instruction would do, but the primitive for "ROTATE" could probably not be done more than ten percent more efficiently if coded in straight assembler.

What exactly is a macro in an interpretive language? Does the inclusion of a macro facility make a language extensible? It should be noted first that the distinction between macros and subroutine calls in an interpretive language is fairly nebulous at times. To answer the above
questions, Stephen Shuman discusses extensible interpretive languages which handle macros much the same way that the Graphics Symbiosis System does. He explains concepts that become evident to anyone who has written or designed a language of this type, but probably to no one else:

The most important advantage accruing from this sort of organization is that it provides a particularly coherent framework for accommodating extensibility. The fundamental concept—that new operations are defined in terms of existing ones—becomes essentially equivalent to the elementary capability for defining procedures (hence, in one sense, an extensible language really requires nothing "new"). From a linguistic point of view, such an approach guarantees the requisite definitional consistency, insuring that the operations introduced by extension are fully compatible with the primitive operations, since both are formulated in terms of the same mechanism.

...The ability to invoke an extended operation...corresponds, in effect to an "interpretive subroutine call". This primitive instruction serves to establish a new level of interpretation...The constituent instructions of this operation (possibly including further subroutine calls) are executed in exactly the same fashion as any other sequence of primitive instructions, until such point as the entire operation has been carried out, whereupon this level is abandoned and control is returned to the program which contained the original call (with the result, if any, being retained in the restored environment.) [18, p.122]

Schuman goes on to describe the overhead of the "cascaded definitional structure" associated with extensible languages [18, p.124] and a separate compilation phase which would eliminate the cascading overhead. The macro facility in the Graphics Symbiosis System at this point is what Schuman would call "traditional" because it makes no attempt
at attacking this problem (some overhead is eliminated by skipping the syntax checking of certain system utility macros, as will be seen later, and arithmetic statements will eventually be handled in a compiled manner, however.) It is not clear at this point that the cascading will reach "impossible" proportions in the graphics context. If and when the macro facility becomes too cumbersome to use with large, multi-level macros, some sort of compiling feature or syntax shortcuts will have to be added. This type of mounting complexity will only rear its head in numerical calculations and not with the more efficient primitives such as "ROTATE," "GROUP," "SMCOTH," "SCALE," "SHADE" and so on. The writing of a special purpose arithmetic compiler is a task that will probably be undertaken in the near future; the need for compiling other commands is not yet proven in this environment.

It should be clear that an interpretive language with a flexible macro facility can be called "extensible." Macros are used to define new syntax meanings in terms of old, that is, to add new primitives to the system. Each user will, no doubt, find a need for his own unique macros and these will extend the system in his direction in particular. Jean Sammet, for one, believes this to be an important feature of such a language: "In fact, it may even be true that the only practical application of extensible languages is to produce specialized application languages and their compilers or
interpreters" [19, p.141].

Very ambitious extensible languages are typically at least as difficult to learn (if not more!) than regular programming languages. The procedures one must follow to extend data types and operators are too often quite complicated and mathematical. Yet, in the case of the Graphics Symbiosis System, the goal is to let the sophisticated user—one with a decent amount of time logged on the system, but who is not necessarily a programmer—write his own extensions to the language. Furthermore, the naive user should be accommodated, that is, the habitability should not be sacrificed for the extensibility. Is this impossible? Does a tradeoff exist between extensibility and habitability? If the methods of achieving extensibility are simple to use then the answer is "no." In other words, to maintain the system's habitability, the macro facility must be habitable as well. As stated in the last chapter, macros are easy to learn to write and easy to use, a fact that is verified by the non-trivial macro writing by non-programmer artists currently using the system (see "DLPCI" in Appendix C). Thus, the system was conceived to include a programming language for non-scientists which would also permit, and hopefully lead the user to a high level of sophisticated programming. Given the advantages of visual feedback and high motivation, the users have shown that the Graphics Symbiosis System can
handle many classes of graphics problems to a very demonstrable practical level.

The Graphics Sybiosis System—like APL—might be called operator-rich but data-poor. The data types in graphics are essentially one-dimensional, two-dimensional and three-dimensional vector lists. This system also has integer, name and device variables, as well as integer constants, register load values, three-by-four matrices and $x,y,z$ coordinates as data types. The sophisticated user can manipulate words, bits, and bytes in any radix, or ASCII characters. The power to programmatically add new data types (e.g. packed decimal) was never considered to be worth the effort of development, since its usefulness has not been shown to satisfaction.

One can, however, change data types as far as the display allows. The Vector General Scope has some twenty data types, of which four are used by the system. A user can, with care, build lists of any peculiar form at the risk of abending the display interrupt routines if he makes a mistake. Naturally, many of the system functions will fail on unanticipated data types, since most of them rely on three-dimensional absolute vector lists.

Defining new operators, however, is a very important feature to a graphics implementation. Let us, then, in the next several pages, build up from the basic command syntax
to the system features which allow the writing of extensions—for it is in these features that the true extensibility lies.

The simplest type of macro is written at asterisk or system command level. It allows delay of execution until a matching bracket is seen. For example:

\[<\text{FLAP WING1,-300,300} \]
\[\text{FLAP WING2,-300,300}>\]

will start the two wings flapping together, since the time lag between processing the first line and the second is small enough to have no visible difference. Chances are that the macro would execute completely before the next display cycle when the changes would take effect, so the instructions appear to be simultaneously executed.

When the command syntax checker sees the left angle bracket as the user types in the code, it immediately builds a special system name (&&SC01 to &&SS99) for the code being typed. It then continues building in the core allocated until a matching angle bracket is found. The user is notified that he is building a macro by the plus sign appearing at the beginning of each new line until the macro is finished. If he should desire, he can nest this kind of macro (i.e. have an unnamed macro within an unnamed macro), although no reason to do this has yet been found in practice. The macro can be up to 4,090 characters long, if
the user has the physical stamina to keep typing that long. As soon as the matching right angle bracket is seen, the code in brackets is immediately executed. When execution is finished, the macro is automatically deleted and the storage it used is reclaimed. All looping facilities (to be discussed later) are allowed, and the user may also call named macros from the disk within this type of macro. These unnamed macros (as they are called) are particularly useful for testing short segments of code, since no subsequent commands are required to execute or delete them. An example created during a film exposure test is shown in Figure 8. It was better, in this case, to write a short macro to lower and raise the landing gear of the airplane than to manipulate the dials—exact reproducibility was required in the test. Since this macro did not have to be saved, it was created as an unnamed macro.
AIRPLANE -- A short macro was written to raise and lower the landing gear of the airplane. This airplane can fly about with the propeller spinning. If the user desires, he can also flap the wings, open and close the cockpit, eject the occupants along a path, and turn the wheels of the landing gear, all at the same time.
Named macros, on the other hand, are for production purposes. One can create this type of macro to eliminate typing fatigue, of course, if the code is to be executed often. But named macros are essentially what allow the line-by-line interpreter of the Graphics Symbiosis System to be transformed into a programming language.

When the command syntax checker sees a colon (":"), it goes into named macro building mode. First the name is built corresponding to the letters preceding the colon, then storage is allocated to the name and the ASCII characters are read into core, in the same way as the unnamed macros are handled. All macros are kept in ASCII format at all times, a technique which permits dynamic changes to occur to code even as it is executing. Syntax checking is made easier since a line is not checked until it is executed. This use of the macro as a text string makes re-entrancy and recursiveness automatically possible since the code is always "pure." The time cost involved in executing macros, of course, is part of the interpretive nature of the system. It is much easier to have a compute-bound macro take noticeable time than an assembler routine which performs the same task. For this reason, complicated system utility macros, like the sonic pen drawing routine, may elect to bypass the syntax checking and reformatting loops, if the author has been careful to test the macro in advance. Such macros must be executed from the common area on which no
user can write without entering the suitable password.

Macros can be created within macros. For example:

TOM:<PROMPT "WHAT IS THE NAME OF THIS PICTURE?"
       INPUT $A ; ASK FOR THE NAME
       GETDSK A
SAM:<MOVE $A,JS ; TRANSLATE THE PICTURE USING
       ; THE JOYSTICK
       SCALE $A,SX ; SIZE VARYING WITH THE X
       ; VALUE OF THE SONIC PEN
       ROTATE $A,X,DO ; ROTATE IT AROUND THE X AXIS
       SETINT $A,D9 ; INTENSITY ON DIAL 9>
       DC SAM>

When the user types "DO TOM," the macro "TOM" will get the
figure whose name is passed by the user response to "INPUT
$A." The picture's name in core is then the same as it was
on the disk. "SAM" is built dynamically, and then executed
when the "DO SAM" line is reached. Later, at any time, one
could type "DO SAM" or just "SAM" and the code in "SAM"
would be re-executed using whatever name the "$A" variable
held at that point. If "TOM" were re-executed, the macro
routine would try to re-build "SAM" and a conflict in naming
would result since the system will not allow
multiply-defined names. Typically, the last line of such a
macro is to delete itself, so no housekeeping would be
necessary to re-execute "TOM." Any other routine that used
"$A" could use "SAM" as a subroutine. Also, "INPUT $A" is a
legal asterisk level instruction which simply takes the next line of input as the name, so that "SAM" could be used at any time if preceded by an "INPUT $A" in any mode. Note that no effort has been made to have local and global declarations of names, and it is not at all clear at this point that this omission could result in deficiencies severe enough to warrant the complexity of permitting declarations of local variables. In fact, allowing local variables makes it difficult to enter macros at points other than the beginning. As will be seen, the present macro structure allows very flexible program flow control in sometimes quite unusual ways. There is no limit to the nesting of macros other than user fatigue and processor stack space on the PDP-11. It requires about eight words of storage in the push down stack to go down a level in a macro—five are for register saving and three for return information. Consequently, at the cost of eight words per call, a macro can reenter itself until the stack underflows. If this happens, the user is notified and returned to asterisk level at which point the stack is reinitialized. Of course, very little can be done recursively in a program that cannot be done in other ways, but at least the capability is there for the use of the clever or lazy user (providing powerful tools for lazy users is a habitability consideration, of course).

The single most significant thing computers do is loop. Looping permits repetitive tasks to be done at electronic
speed, ad infinitum. Looping is essentially what distinguishes hand calculators from computers. Accordingly, a graphics language should have simple as well as sophisticated looping facilities. These facilities are known as the program flow control commands.

"SKIP" is the simplest of these commands. The operand that "SKIP" takes indicates the number of lines forward or backward the control is to be passed. A skip with a negative number transfers control backwards and is used mostly for looping. A negative number exceeding the number of lines preceding the "SKIP" in the macro will result in transfer to the first statement. A "SKIP 0" will lock into a loop, so it is not allowed except if it is part of an "IF" statement, which is discussed later. A "SKIP" with a positive operand will skip over the number of lines indicated. Since a line is not syntax checked unless executed, a "SKIP" could be used to branch around some non-executable alphanumeric strings.

Furthermore, the operand to "SKIP" can be a full expression. Thus, "SKIP A" will result in a skip of the number of lines indicated by the numerical value of the variable "A." Since expressions are evaluated dynamically, the program control can be the result of a computation. "SKIP D0/2-200" is also permissible, although of questionable utility, since it causes control to skip the
number of lines indicated by half the value of dial 0, whatever it might be at that time, minus two hundred.

To eliminate counting fatigue in compiling skip offsets, disk-resident macros may have pseudo-labels for "SKIP" operands. When these macros are called into core, the pseudo-labels are resolved and replaced by numeric values much like an assembler resolves addresses. One may create a macro with pseudo-labels in the ordinary manner, but he must "COMPILE" it to resolve the labels. If he neglects this step, he will be notified by an error message that unresolved labels exist. Pseudo labels are preceded by a percent sign (%).

Named macros are called into execution in several ways. For system utility macros (see Appendix C), "CALL macro-name" is sufficient. The "CALL" processor first checks to see if the macro is in core, and if it is not, it is called in, and then executed. Much of the syntax checking is disabled if "CALL" is used. If the user knows that the macro is already in core, he can simply type the macro's name to cause execution. Most system utility macros delete themselves upon exit, however, to reclaim space, so the use of the "CALL" is usually necessary each time.

The user also has the option of getting all his macros into core at once by using "GETDSK" or "GETCOM" commands. Then, when the macros are called, no delay attributable to
disk access overhead could ever be present.

If the macro is on his own disk area, that is, if the user is the author, a "DO Macro-name" will get the macro for him. Syntax checking is not disabled, however, so his macro will run slightly slower than a comparable system utility macro. If he includes the disk area specification, he can use another user's macros. For example, "DO FLUFF[30,11]" will allow access to the macro "FLUFF" on area "30,11." Similarly, "DO JSDRAW[31,1]" would get the system utility joystick draw routine but enforce syntax checking (31,1 is the common area). The clever user could "GETDSK" a macro from any area on the disk, and then "CALL" it, thus eliminating syntax checking, at his own risk, of course.

Again, the macro calling facility is very flexible, but all the ordinary user need know is the name of the macro he wants and the simple syntax of "CALL" and "DO."

Although it seems like the discussion has strayed from the program control features, it really has not. "CALL" and "DO" also allow subroutine linkage to modules which are either disk or core resident. Typing just the macro name without "DO" or "CALL" provides linkage to core resident modules only. There is a further operator "GOTO" which acts like a jump rather than a subroutine call. "GOTC" takes a macro name and an optional positive displacement, and will transfer control to that macro plus the displacement in
lines, if any. Of course, if the calling routine were in turn called by another routine, the return is to the original calling routine.

For very complex nesting within macros, the user has to keep track of where he is—one could hardly expect the machine to second-guess him. Presumably, the user who can think recursively and program nested subroutine and macro calls is competent enough to keep track of them. Commands like "LIST" and "PROMPT" can help in debugging macro code, however.

The "RETURN" command will pop up one level in the macro nesting. It is equivalent to a "GOTO" to a no-operation macro. If it has a positive operand, "RETURN" will pop up that many levels of nesting. The "EXIT" command is equivalent to a "RETURN" whose operand exceeds numerically the level of nesting. Thus, "EXIT" always returns one to asterisk level.

All these program flow control features are nearly useless without conditional statements, however. The Graphics Symbiosis System has very flexible conditional statements too. There are synchronous and asynchronous conditionals. The synchronous conditionals are triggered by the "IF" operator, and can be set up and used dynamically on any level. The asynchronous conditionals —called "ON" conditions after "PL/I"—may be set up in a macro, but are
only tested and executed during command input wait at asterisk level. At this time, the command level processor is idle and can conveniently process such requests if the conditions are satisfied. "IF" statements must be continuously looped upon, or "polled" in a macro if the condition is to be checked at any time. Therefore, a macro must have control of the system if an "IF" statement is to have any effect, so it is only within macros that synchronous conditionals make sense. "ON" conditions, on the other hand, are for calling macros when no macro has control. The distinction should become clearer as the discussion progresses.

The syntax of "IF" statements is as follows:

IF FSn operator expression, simple command

or

IF DEV operator expression, simple command

"FSn" is one of the sixteen function switches, or if specified as "CPn," is one of the sixteen PDF-11 console switches. "Operator" is one of the usual arithmetic comparison mnemonics: LT, GT, EQ, NE, GE, and LE with "=" and "#" also substituting for "equals" and "not equals" respectively. "Expression" is an combination of numbers, variables, dials and arithmetic operators of the type previously discussed. The comma terminates the clause, and if the condition is satisfied, the simple command which follows will be executed. A simple command is defined as
one which will fit on the rest of the line, a hardware dictated restriction that can be most simply circumvented by use of the "DO" operator, as described below. "DEV's" may be also tested with "IF" statements as the second syntax specification shows.

If the condition is not satisfied, control passes to the next statement in line. Thus, "IF FS1=0, SKIP -10" will cause the code to loop ten instructions until FS1 is pushed. With function switches, if the operator and expression is left out, "=1" is assumed. "IF A LT 2000, DO GEORGE" will cause execution of the macro "GEORGE" if the variable "A" is less than 2000 at the point of execution of the "IF" statement. Similarly, "IF K GT M+N+2000, PROMPT "K IS TOO BIG" will send the message to the user if the condition is satisfied. If a multi-line sequence must be executed as a result of the "IF" condition being satisfied, the user has the choice of creating a named macro to do the sequence and then executing it by using a "DO NAME" as the simple command, or setting up the complementary condition to skip around the multi-line sequence that would then follow the "IF" statement. The former approach saves storage because the multi-line sequence may be disk resident until needed, and can delete itself upon completion. "COTO's" and "RETURN's" may be used for simple commands, as well as arithmetic statements, or any other command (except "TEXT," which is multi-line). Furthermore, as long as it will fit
on one line, several "IF" statements can form a nested conditional in a rather elegant way:

    IF FS1, IF A LT 20, IF FS2 NE 0, IF DO GT 0, HELP

A user could build a macro as long as it fit on one line:

    IF FS1, SAM:<FILL TOM,FRED,HAIR,D7>

although this is somewhat limited in application.

The asynchronous conditions are treated differently. The syntax is as follows:

    ON DEV operator expression, on command
    or           ON FSn operator expression, on command

The operator and expression specifications are the same as for the "IF" statement; however, following the comma, there must be either a macro name preceded by "DO" or "CALL" or a macro building sequence. The difference here is because the table of dispatch addresses for the "ON" condition processor must have an address of macro code to correspond to the condition. The user may also specify a named or unnamed macro building procedure in the normal way. If named, it is equivalent to building a macro and then using an "IF..., DO..." sequence. If unnamed, however, the macro will be automatically deleted after the condition is satisfied and the code executed. Thus the following are possible:

    ON FS1, <DELETE SAM>

    ON FS15, <ROTATE GLOBE,X,DO,D1,D2
              SCALE GLOBE,D9>
A macro could easily be written to set up all the "ON" conditions that a user might need for himself or his second-level user. Then, anytime such a condition was satisfied the appropriate action would be taken and the condition disabled. If continuous monitoring is desired, the last statement of the code executed as a result of the "ON" condition could set the condition up again. Typically, the macro author would turn off function switches corresponding to those which triggered execution in this case, or the condition would be constantly satisfied. Since the macro executed as a result of the "ON" statement could have "IF" statements in it, logical "and" and "not" conditions may be programmed by checking more function switches or "DEV"s." Of course, since the very nature of the "ON" condition is a logical "or" operation, a logically complete set of interactions with any device, variable, or function switch collection may thus be realized. Furthermore, since coordinates may be read into variables by the "GETPOIN" command, elementary data analysis may be undertaken by a series of "ON" and "IF" conditions.

"ON" conditions, as was stated before, are only checked while the system is waiting for a command. If a macro is in progress, the macro author can issue a "WAIT" command to force an input wait if necessary. One or all of these "ON" conditions could trigger code containing a "RESUME" command which would return the user to macro control. The tactile
interaction can thus be as simple or complex as the user desires, an option not usually offered in a programming language.

Keyboard substring matching is a special extension to the "ON" and "IF" conditions. This feature allows very flexible decision making on the basis of an anticipated set of keyboard inputs. The macro author sets up his "KB" conditions as follows:

IF KB=text string, simple command

or

ON KB=text string, on command

where "simple command" and "on command" were described before. The user has the option of specifying a modifier of "KB" such as "KB5" which would match only the fifth word in the line, with normal English punctuation taken as word delimiters. In the case of the "ON" condition here, once it is enabled, every line input on the display terminal is checked for matches. If the input line does not have a match, the line is passed to the line processor to be executed as a command. If the line is not in valid command syntax, of course, an error message will be sent to indicate the fact.

The "IF KB" condition is, as might be expected, used within macros. Its use follows an "INPUT KB" command which calls for a line of input which is stored in a special line buffer. The desired strings can be tested by looping the
"IF KB" statements. The line remains in the buffer until another "INPUT" is issued.

As the need arises, a more "SNOBOL"-like string matching facility could be developed, using these conditions as a framework. A universal matching character could be added to allow matches of strings that are less than or more than one word. Since the keyboard match tables are built dynamically, the user can set up many entries before running out of table space. At this point, the Graphics Symbiosis System is still too new and the users have not begun to write macros for "third-level" users. When more sophisticated pattern matching is required, the code can be expanded fairly easily.

It should be realized by now that the macro facility allows very powerful extensions to the language to be written in relatively short amounts of time. Thus, an instructor could appear one day, have an experienced user write a few macros which define a special system, and fairly complex sequences could be developed, filmed and sent off to be developed. An educational movie could be produced in a matter of days, if necessary, so an instructor could conceivably even use the system to generate film to explain some concepts he found difficult to explain conventionally earlier in the week. The language of the Graphics Symbiosis System can be quickly tailored to the needs of any
particular naive user, through the power of the macro facility.

Previous computer animation systems have at most a small fraction of this power. Most graphics efforts have, in fact, consisted of linking a series of one-shot, throw-away, single purpose programs not by subroutine linkage, or under a system of any sort, but in a film lab, where the film clips are assembled. In this light, the flexibility of the Graphics Symbiosis System is probably best appreciated.

Occasionally, there will be features that a particular user may want to add that simply cannot be done in a macro. Macros do not as yet have matrix multiplying capability or floating point variables. Also, if speed is all-important, the user may want to have his algorithm coded in assembler rather than in a macro. The Graphics Symbiosis System is designed to make adding assembler modules very easy. A new name can be generated, added to the system function table, and the interfaces to allocation, expression evaluators and names accessing routines can be added as a series of subroutine jumps in a matter of five or ten minutes. The user has only to code the heart of the algorithm.

If the system were to continually grow as the result of adding new assembler routines, it would be at the cost of picture space. The present memory limit of the system is
28k words. Fortunately, a method was planned and recently has been implemented to allow modules of assembler code to swap from the disk providing certain conventions are observed. Access to system resident modules is through a fixed location table, but the swapped module may be allocated the smallest chunk of core in which it will fit, and thus be anywhere in core. Fortunately, the PDF-11 assembler creates position-independent code unless it is specifically subverted, and this, of course, makes preparing swapping modules much easier. About half the system code is now non-resident. The system has yet to require any major modifications to implement new assembler algorithms, and should be able to accept virtually any suitable code as its own.

The programmer who desires to add an assembler swap module must first modify the system names table to include the new name. The dispatch address of the new module becomes -10 which indicates that it is a swap module on the disk. He then codes the module, including the bit 12 lockout convention for functions in the first word and -10 in the second word. (Control is always transferred to two words past the beginning of a module.) The module must exit with a standard subroutine return. If the programmer wishes to communicate with system modules (which he no doubt does), the LNKSUB assembler macro sets up the proper linkage. Register four, if used, must be restored before entering the
LNKSUB expansion since it holds the base address of the LNKSUB dispatch table. The new module is assembled with the system parameter file S.MAC. The object module is then linked with SWPMAK.OBJ and SWPEND.OBJ, the first of which later causes the resulting load module to write out the new module on the disk in the proper form for a swap module. SWPEND is simply an end directive. The load module is then executed and produces a file SWP.TMP which must be renamed to the intended name. This procedure normally takes only a few minutes once the algorithm is coded and typed.

Clearly, a language which depends solely on adding assembler routines to expand its capabilities cannot be described as "extensible." Note, however, that extensible languages became extensible only by adding enough assembler routines to make the language rich enough to define new syntax meanings in terms of existing ones. The aim of the Graphics Symbiosis System was always to provide a viable system whose obsolescence was neither envisioned nor planned, at least until the technology changes. It seems that, for the moment anyway, the data manipulation part of computer graphics is under control to a very usable degree. This system should remain a creative tool for a few years, which is about the best any new programming effort could expect.
Chapter III—Implementation Details

Don Knuth's "Art of Computer Programming" volumes are a notable exception to standard computer science works in which all too often the capabilities of a system are given, but not the methodology involved. Knuth supplies information of the kind normally available only to readers of microfiche. Following Knuth's example, this chapter will give some insights into the internal workings of the algorithms that make the Graphics Symbiosis System work. Considerable time was spent generating these methods, and if they have been duplicated elsewhere, it is not the intention here to assume uniqueness. Each routine was worked out independently, and not consciously adapted from any other source.

The Graphics Symbiosis System was inspired greatly by the study of and workings with the PDP-10 time-sharing monitor, TECC—a text editor of great power (also on the PDP-10), and a couple of years of interaction with System/360 Operating System and Job Control Language. The last of these inspirations was mostly in a negative sense—no graphics language could ever look like JCL and be usable by artists. A goal, however, was to provide the kind of flexibility JCL offers—so one could, among other things, spend most of his time writing macros, as many systems programmers do with JCL. Some of the advantages of BASIC,
PL/I, ALGOL and SNCBOL were studied in preparation for the project. And, of course, many graphics systems were consulted, although all of these were painfully lacking in implementation details. This chapter is for the reader looking for implementation ideas, but who does not enjoy deciphering assembler code.

A. Reasons for the Assembler Implementation

One of the unconventional aspects of the Graphics Symbiosis System when compared to other graphics efforts, is its assembler implementation. Most graphics systems use FORTRAN as a host language—but most systems do not contain extensible programming languages nor do they have the kind of intricate interrupt level structures found in the Graphics Symbiosis System.

The reasons for rejecting FORTRAN are not unique to the graphics environment. Computer operating systems are not written in FORTRAN and no usable implementation of SNOBOL, COBOL, or ALGOL exists as series of FORTRAN-callable subroutines. Jean Sammet states that standard FORTRAN has never been used to bootstrap itself [24, p.150], a comment on its lack of extensibility powers. FORTRAN was not intended to do much more than translate formulas into machine code: "Its largest disadvantages stem from attempts to use it outside of the realm for which it was intended, namely for any type of alphanumeric data handling" [24,
Languages which have adequate character handling and bit manipulation (PL/I, BLISS[25]) are not widely available, especially on mini-computers. Since FORTRAN is so awkward in the operations necessary for language processing, any system written in FORTRAN cannot do significant input string parsing, and be even remotely efficient.

FORTRAN's bit manipulation deficiencies would also make it difficult to attain much sophistication in the storage allocation and reclamation routines and the traversal algorithms to be described in this chapter. Lack of pointer variables makes list processing unnatural, to say the least: "Because it was designed so early, better ways have been found to do almost everything that is currently in FORTRAN" [24, p.169].

If PL/I or BLISS were available on mini-computers and had reasonable memory requirements for operation, one could consider using a higher-level language to implement a graphics system. One cannot, however, justify implementing BLISS or PL/I on a mini-computer to save oneself the effort of coding in assembler.

FORTRAN is available but it is not acceptable for large amounts of character manipulation. Why is character manipulation so important to animation graphics? First, to deal with the complex sequences often required, a full-scale
programming language is needed, that is, no practical series of button pushes or intensity of lightpen waving by itself will ever convey the amount of information that a programming language can. Of course, one cannot write macros solely in terms of button pushing either. The reason why this necessary programming language cannot be FORTRAN itself (as a language) is the second major reason for creating a new language: interactive graphics requires an interactive language. Interactive languages are interpretive languages, which FORTRAN is not. Therefore, one is left with the task of writing an interpretive language in FORTRAN—precisely why character manipulation is such an issue.

It may not be obvious to batch computer users why interaction is important. Thomas Standish gives two good reasons:

The first is that conversational systems which provide rapid response to trivial requests permit one to evolve and modify a program at a very rapid pace.... Conversational systems permit you to observe the misbehavior, to request a trivial change and have it rapidly satisfied, and immediately to set the program in motion again to check whether the new behavior is correct. One should not have to go through the whole cycle of updating a source file, compiling, loading, initializing and then setting up the proper routine circumstances in order to sense the behavioral implications of a trivial change. The second case in which interactive computer animation systems are important is to satisfy applications where man must be in the loop to supply decisions about how the animation is to proceed....man must set the time scale, the focus and the level of detail. Since man has
changing interests that the machine can't predict, he must be in the loop to decide what to look at, to compress the uninteresting, to expand the interesting, and to jump in and out of the hierarchy of detail depending on whether he is excited or bored. Conversational graphics systems and conversational programming languages are structured to permit just this sort of thing. [26, p.308]

Non-interactive graphics, such as plotting and microfilm recording can be done well in FORTRAN. However, one cannot simply type "CALL ROTATE (BIRD)" in FORTRAN syntax and have the bird instantly rotate on the screen, given the subroutine and picture already exist. One cannot even hand the computer a card out of a deck—FORTRAN programs, even single statement ones, must be compiled, loaded and executed—and this is not particularly interactive or desirable. To interpret "CALL RCTATE(BIRD)" takes as much work as "RCTATE BIRD" if not more, so why maintain the FORTRAN syntax?

The advantages of FORTRAN outweigh its disadvantages in some applications of graphics. For instance, users of microfilm plotter packages such as the Polygraphics Package [41] often take advantage of the many subroutines which exist in FORTRAN submitted by other users from all over the world. Furthermore, FORTRAN library subroutines to solve systems of equations, integrate and differentiate functions and perform various statistical and mathematical tasks are readily available, if necessary. Of course, rapid response to trivial changes cannot be obtained with this approach
since everything has to be edited, re-compiled, linked and re-executed to affect each change in the output. Thus, even if a user can get feedback from such a device in ten seconds, it might take him twenty minutes to make a simple change. Until an industry standard graphics programming language is adopted, the user will simply have to choose between interaction with interpretive languages and the kind of exportability and availability of libraries of subroutines represented by the FORTRAN approach. Of course, it would be possible to write an interpretive language for a microfilm system which would eliminate the compiling and loading problems but then the advantages of FORTRAN would no longer be available.

Similar arguments may be raised for FORTRAN implementation of storage tube systems and the same advantages and disadvantages exist. The storage tube has a better response time than a microfilm plotter (one-half second with a high-speed line), and is a good alternative when equipment like the Vector General is not available. If the system designer is more interested in using previously coded algorithms for such things as hidden-line removal, shading and character generation, he would be advised to use FORTRAN. If he is more interested in interaction and habitability, he will probably have to write an interpreter and recode the algorithms he wants by himself, in assembler or another host language.
In any graphics implementation, several routines would have to be written in assembler no matter what the base language. Interrupt level code and vector list manipulating is best done in assembler, since higher level languages do not typically have interrupt trap facilities or data types of the vector relative variety. In retrospect, the only algorithms that could have been more easily coded in FORTRAN are the SMOOTH and SHADE routines, and the effort saved would not justify putting up with the compiler.

Higher level programming languages are designed to save programming time. The amount of time it would take to add enough bit manipulation and character handling facilities to FORTRAN would exceed any time saved by coding in FORTRAN, and the end product would probably run slower (compilers with optimizing options like IBM's FORTRAN H are not available on mini-computers either). Any hopes of machine independence would be lost also, and machine independence is one of the reasons people use FORTRAN in the first place. A FORTRAN-based system cannot achieve the level of flexibility possible with an assembler implemented system. Animation graphics is the type of endeavor that demands all the computational speed and flexibility it can get, and then some.

B. Storage Allocation and Reclamation
Storage is allocated in chunks of thirty-two word blocks, up to 128 blocks contiguously, or 4K of memory. These sizes were chosen to give some degree of compatibility with the segmentation hardware of the PDP-11/45, so the system could be upgraded to a multi-user re-entrant system eventually. One of several allocation methods could be used in a graphics language. A very simple allocation scheme would chip off pieces from the large chunk of available core, until there was none left. Then the user would have to restart the system, there being little effective way to reclaim storage.

If larger blocks of core were chosen as the minimum size—say 512 words—a bit map with ones corresponding to open locations could be used for storage allocation and reclamation. When a block was deleted, the corresponding bit would be turned on again. To search for the largest contiguous chunks would be fairly costly with this scheme, however, especially if smaller block sizes are used.

The allocation block size was chosen partially to keep the addresses of allocated elements on thirty-two word boundaries to conform with the segmentation hardware. A smaller block size would be impractical anyway, since only a trivial macro could fit in sixteen words. As it is, two blocks are required for even the smallest picture since the size of the register load block exceeds thirty-two words.
Larger size blocks were not used because the amount of storage wasted by rounding off the number of words actually used to the next block increases as the block size gets larger. No simulation studies were undertaken to ascertain the sagacity of the choice, but the thirty-two word block has proven to be quite acceptable nevertheless.

Another approach to storage reclamation is by shuffling as the non-paging PDP-10 does it. Shuffling dynamically compacts core by re-adjusting addresses, and thus leaves all the free space at the top. However, in a graphics system, it is not really possible to shuffle while maintaining a real-time display. Even on the PDP-10, display jobs are typically locked in core and never shuffled, unless the display has finished, and this has even been known to confuse the system more often than not. The advantage of shuffling is that one can maintain all the unallocated core in one large chunk, if necessary to do so. However, if the display can accept either hardware or software jumps to continue vector lists, one can really use core as if it were all in one big chunk anyway.

The Graphics Symbiosis System makes major use of push down stacks, as will be seen. Allocation is handled by placing address/displacement pairs on a separate allocation stack. The displacement is a count of thirty-two word blocks which ranges from one to one hundred twenty-eight.
The address is the physical core address, that together with the displacement should not cross a 4K boundary—again a consideration for segmentation and paging.

Two types of allocation requests are honored: a known-size and an unknown-size request. The former is used when disk resident files whose length is known are transferred into core. The latter is needed during dynamic creation—that is, drawing, macro creating, and any other data generating processes like "SHADE," "FILL," and "PUTFOI" (see Appendix A for explanations of these commands). Each time allocation is called, a chunk of core is passed to the calling routine which uses what it needs and returns the leftover blocks. The address and displacement of the leftover chunk are pushed on the stack and a sort is done on the address/displacement pairs based on the displacements with the largest elements going to the top. Since the bottom of the stack is always known, the known-size allocation routine starts at the bottom and compares upwards until it finds a large enough chunk (remember that the smaller chunks are nearer the bottom now). If there is no chunk large enough, the largest is taken, and then dynamically linked to another non-contiguous chunk when necessary. Thus the best fit is chosen for the known-size case.

Again, no simulation studies were undertaken to
determine if the best fit technique is, in fact, best. A justification for the choice of method can be given, however: it is imperative that the largest possible chunks of core be maintained, and that small pieces not be constantly chipped off the largest blocks, but fit into small pieces if possible. For example, the character set subroutine blocks for "TEXT" vectors cannot be continued and chained to other blocks, and many are nearly 4K in size. If many small elements were allocated by first fit, it is more likely that the large chunks would not be available when needed. This argument is less effective if the allocation stack is never sorted, and in retrospect, a slightly more efficient allocation might be obtained if the algorithm were changed to eliminate sorting and best-fit usage. In any case, a trivial change in the code could realize these modifications if the effort was deemed necessary. At the time, it seemed the approach taken would assure efficient utilization of core in the short as well as long term case. Arguments for the first fit method usually only consider the long term case, which the naive user may just never encounter.

When the size is unknown, the allocation algorithm takes the top element from the stack since it is guaranteed to be the largest. If that chunk turns out to be too small, the next largest chunk is dynamically appended to it by a jump and the process is repeated until enough core is gotten
or all the core is exhausted. Of all the functions in the system, "SHADE" and "FILL" are the best at running out of core—it takes a lot of coordinates to transform a wireframe three-dimensional drawing into a half-tone representation.

A word might be said about the sorting of the allocation stack, which seems to offend the sensibilities of those who find sorting unacceptably inefficient. First, the stack never has many elements because they are combined when possible, so that even if the sorting takes time on the order of the square of the number of elements, the time taken is still trivial, especially since allocation is not done as often as the traversal algorithm, for example. Secondly, sorting the stack made implementation easier in that it was simpler to check if core was allocating and deleting properly, and all blocks of core could be tested for position independence of pictures and algorithms, without having to fill all of core each time. Since development is still in progress, these allocation techniques have not been abandoned. Furthermore, to use a best fit technique, some sorting has to be done anyway, and criticism of sorting is only valid if the number of elements to be sorted is considerably larger than ever found on the allocation push down stack.

The "DELETE" command reclaims storage and returns it to the allocation stack. First, the displacement and address
of the deleted item is placed on the stack. Then, each element down in the stack is tested to see if it is physically adjacent above or below the new element and joined if it is. The element physically above the new element is called the "physical preceder" and the element after is called the "physical follower". An element can have at most one physical follower and one physical preceder on the stack at a time. The testing procedure can now be explained. First, the newly pushed displacement is added to its accompanying address, and the sum is saved. This number corresponds to the physical follower. The stack, which typically has less than ten entries, is quickly searched to see if the physical follower appears there. If it is, the displacements are added, the first address is kept, and the physical follower address/displacement pair is compacted out of the stack. If the physical follower does not appear on the stack, it simply means that the core following the new element is still allocated. Second, the address/displacement pairs on the stack are each added to see if the sum matches the address of the new element. If so, the old address is kept, the displacements added, and the top two words—the new element's address/displacement pair—are popped off. A sort then arranges the displacements in descending order and the algorithm is finished. Thus, the largest possible chunks of storage are maintained without shuffling. All this housekeeping is
transparent to the user, and takes much less time than a carriage return.

The storage allocation and reclamation routine is very heavily used in the system, and is, after much effort, a solid and dependable piece of code that is "user-proof." Macros, swapping assembler routines, pictures and group nodes all use this routine.

C. The Line and Macro Processor.

The line processor handles the interpretive part of the system. It includes all the syntax checking that ever goes on, dispatches to the routine to execute system functions (like "ROTATE," "GROUP," "GETDSK," etc.), builds and executes macros, and calls the arithmetic routines. The notable part of this complex routine is the way in which it handles nesting, reentrancy, and recursiveness.

Several key characters trigger special processing. A colon indicates that a named macro is about to be built. A right angle bracket marks the creation of an unnamed macro. An equals sign calls the assignment statement routine. In addition, several special words are checked for: "CALL," "DO," "IF," and "ON," since these operators may require operands which violate normal line syntax.

The named-macro processor dynamically calls allocation for a chunk of memory, at which time it records the ASCII
text string into core until the last matching angle bracket is seen. The name is built in the normal way, using the letters preceding the colon, and the user can refer to his macro, delete it, save it on the disk, or change it. The named-macro processor then returns control to the calling module.

The unnamed-macro processor builds a system name (&&SCC0 to &&S999), gets core, and builds the macro until a matching right angle bracket is seen. This processor retains control, however, and causes automatic execution of the unnamed macro by pushing its own status on the stack and doing a re-entrant call to the line processor. Thus, if nested angle brackets are seen, a new system name is used and this macro is executed similarly by calling the line processor again. Once an unnamed macro is finished, it pops its status from the stack and resumes the calling routine if any. If a colon is seen, the named-macro processor is called automatically as a result of the line-by-line execution of the macro by the line processor. Finally, when the original unnamed macro is finished, the unnamed-macro processor deletes all names beginning with "&&S" prefixes, thus reclaiming all the space used in creating the macro.

Named macros similarly can nest named macros and include unnamed macros. The line processor simply reenters itself until the lowest level is reached, and then pops its
way back up until back to asterisk level. Note that named macros must be explicitly executed and deleted, and are the only ones which can be saved on the disk.

The line processor also includes the ON processor whose syntax requirements necessitate calling the named- or unnamed-macro processor or the DO processor. The address passed by one of the three calls represents a macro and is built into a table of addresses that corresponds to the conditions. The "ON" condition table—ONTAB (see Figure 9)—contains three sixteen-bit words per entry.
"ON" CONDITIONS TABLE 16 ENTRIES

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>=</td>
<td>DEV</td>
<td>KB</td>
<td>FS</td>
<td>INDEX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16 BIT VALUE OR POINTER IF "CN KB"

ADDRESS OF MACRO TO BE EXECUTED

INDEX = \{ FUNCTION SWITCH NUMBER IF FS
DEVICE TABLE DISPLACEMENT IF DEV
WORD POSITION NUMBER IF KB

BIT 8 → FUNCTION SWITCH ON CONDITION
BIT 9 → KEYBOARD MATCH ON CONDITION
BIT 10 → DEVICE ON CONDITION

BITS 11, 12, 13, AS OCTAL NUMBER
0 → NOT EQUALS
1 → EQUALS
2 → GREATER THAN
3 → GREATER THAN OR EQUALS
4 → LESS THAN
5 → LESS THAN OR EQUALS

Figure 9—ON Conditions Table
The first word indicates the type of device being tested—function switch, dial or variable, or keyboard substring—in the high order byte of the first word, and the number of the device (i.e. ten for "D10") in the low order byte. Also kept in the high order byte are the three bits which indicate the operator, specifically LT, GT, EQ, NE, GE, and LE. The second word contains the sixteen bit value of the expression or device being tested, except in the case of keyboard matching, in which an address of the test string is found. The third word then contains the address of the macro to be executed if the conditions are satisfied.

The DO processor first checks to see if the macro is already in core. If it is, the code immediately passes control to it, pushing the status onto the stack in case that macro has any nesting or "DO" statements in it. Any macro can be made recursive by simply including a "DC" with its own name internal to it. If the macro runs too long without satisfying an exit condition, too much is pushed on the stack and the macro aborts with an error message, but this only happens after several hundred levels. Similarly, macros can call each other until a exit condition is reached.

The CALL processor is like the DO processor except that it changes the disk area to [31,1] and disables syntax checking. Otherwise, they use the same basic routines, so
that a called macro can call or do another macro.

The IF processor simply evaluates the conditions at that instant of execution and if satisfied, shuffles the rest of the line to the beginning of the command buffer and re-enters the line processor. Otherwise, the rest of the line is ignored and control passes to the next line in sequence. Thus, the following is perfectly permissible:

IF FS1, IF FS2, IF A LT 0, IF DO GT 400, SKIP -12

The error message processor interacts with the line processor also. It handles the "creative" error messages discussed earlier. The error message processor can function because the line processor cleverly saves the stack address of the top element upon entry, so that all that has to be done when an error is encountered, is to restore the stack to the saved value. All extraneous elements left on the stack by PDP-11 assembler subroutine jumps and parameter passing are cleaned up in this way. (The central hardware processor also uses the stack for register saving and linkage.) The error message processor can thus recover quite nicely, no matter how far down in a macro the error occurred. The user can then enter as many lines of code as he wishes to be executed before continuing in the macro. He may also just type "^C" or "EXIT" or simply hit the panic button if he wishes to return to asterisk level. If an error were to occur in one of the new lines typed in, the error message routine would re-enter itself and allow the
user to fix the error.

It is hard to believe that some computer systems function at all without hardware push down stacks.

D. Data Structure Traversal

The data structure traversal algorithm runs asynchronously with the rest of the system. It is called at interrupt level by the Vector General display driver. The driver allows PDP-11 assembler routines to be processed between display list calls. Every time the driver sees a final display subroutine return (in Vector General code), it calls the traversal algorithm for a pointer to a new list. The traversal algorithm does more than set up pointers, however.

At the beginning of every data list, there are twenty-eight words of non-display information and another twenty-three words of non-vector drawing display register information (see Figure 10 in which the addresses are given in octal and represent byte offsets—remember, there are two bytes to a word).
## Leaf and Node Information

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>72</td>
</tr>
<tr>
<td>2</td>
<td>74</td>
</tr>
<tr>
<td>4</td>
<td>76</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>102</td>
</tr>
<tr>
<td>12</td>
<td>104</td>
</tr>
<tr>
<td>14</td>
<td>106</td>
</tr>
<tr>
<td>16</td>
<td>110</td>
</tr>
<tr>
<td>20</td>
<td>112</td>
</tr>
<tr>
<td>22</td>
<td>114</td>
</tr>
<tr>
<td>24</td>
<td>116</td>
</tr>
<tr>
<td>26</td>
<td>120</td>
</tr>
<tr>
<td>30</td>
<td>122</td>
</tr>
<tr>
<td>32</td>
<td>124</td>
</tr>
<tr>
<td>34</td>
<td>126</td>
</tr>
<tr>
<td>36</td>
<td>130</td>
</tr>
<tr>
<td>40</td>
<td>132</td>
</tr>
<tr>
<td>42</td>
<td>134</td>
</tr>
<tr>
<td>44</td>
<td>136</td>
</tr>
<tr>
<td>46</td>
<td>140</td>
</tr>
<tr>
<td>50</td>
<td>142</td>
</tr>
<tr>
<td>52</td>
<td>144</td>
</tr>
<tr>
<td>54</td>
<td>146</td>
</tr>
<tr>
<td>56</td>
<td>150</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>62</td>
<td>IF LEAF, FIRST VECTOR DRAW</td>
</tr>
<tr>
<td>64</td>
<td>INSTRUCTION, IF GROUP, WORD</td>
</tr>
<tr>
<td>66</td>
<td>0 IS 0 OR NEGATIVE, WORD 146</td>
</tr>
<tr>
<td>70</td>
<td>IS 0, AND WORD 150 IS THE</td>
</tr>
</tbody>
</table>

- **Leaf and Node Information**

- **Load Rotation Matrix**

- **Figure 10** — Leaf and Node Information
The first twenty-eight (indicated by octal byte addresses 0 through 66 in Figure 10) give all the information this picture needs to generate all the display register settings contained in the twenty-three words (indicated by octal byte addresses 70 through 144 in Figure 10). Every group node also contains the same information. When the traversal algorithm returns control to the driver, it passes the pointer to the first of the twenty-three words which are display register load instructions. The traversal algorithm essentially is occupied with placing the correct values into the operands of these load instructions.

The first word of the picture (indicated by "0" in Figure 10), which is, incidentally, the address corresponding to the name of the picture, is critically important. If this word is positive, and non-zero, the element is a picture (called a "leaf" of the structure). The high order bits give the first trap bits which, if on, indicate rotation, scaling, translation, and path following. Bit 12 is a lockout bit which prevents functions from being displayed and pictures from being executed. Bit 14 prevents any modifications from being recalculated if the leaf is part of a modified group structure (more on this later). The low order bits give the length of the leaf in thirty-two word blocks. Bit 15, of course, is the sign bit. If the first word is zero, it is a group node with no modifications, in other words, a group structure that is
grouped for convenience of reference, or one to which group transformations have not yet been applied. Once a group is used for compounding transformations, it is set to have a negative first word, with the high order bits and all other information corresponding to the leaf case.

The second word (indicated by octal address "2" in Figure 10) of the leaf or group node is a pointer. In the group case, it points down to a lower group node or leaf, or itself if all its sub-elements have been deleted. In a leaf, this word points at the sibling, be it a leaf or group node, or up at a group. The group has its sibling or up pointer fifty-two words down (octal address 150).
Other types of data structures for graphics are discussed by Gray [33], Van Dam [35], Abrams [36] and Williams [37].

The rest of the first twenty-eight words hold ten words of rotation information, two of scaling information, six of motion information, one for cutoff plane and intensity and another one and a half words of trap and flag bits which control the interrupt level algorithms for each group or leaf. There are also words for linear interpolation information plus a couple of words for expansion. The position of this information is identical for group nodes and leaves.

The twenty-three words of load register instructions are executed by the display in the leaf case but not in the group case, where they are merely used for storage of intermediate compounded values. The format was kept identical for the two cases so that the same algorithms could be used to set up both groups and leaves.

The traversal algorithm also makes heavy use of the PDP-11 system stack. (Figure 11 should be consulted at this point to aid understanding of the data structure design.) When a group with trap bits set is seen, it does the modifications specified, records the values in the load register operands, and then pushes a marker followed by the group node address on the stack. The next item element down
(1) BASIC INITIAL DATA STRUCTURE

PATRIARCH NODE
Ø SIGNIFIES GROUP NODE - NO MODIFICATION

POINTER TO LOWER NODE OR LEAF (INITIAL CASE)

END OF DATA STRUCTURE TAGS

(2) BASIC DATA STRUCTURE WITH TWO LEAVES (PICTURES)

PATRIARCH NODE

LEAF FIRST WORD

LEAF FORWARD POINTER

LEAF PICTURE RELATED DATA & VECTOR DRAWING INSTRUCTIONS

TRAVERSAL OF DATA STRUCTURE IS DONE AT INTERRUPT LEVEL AND SERVES TO PERMIT ALTERATION OF DISPLAY REGISTERS DYNAMICALLY AS WELL AS FEED PICTURE DATA TO THE DISPLAY PROCESSOR

Figure 11—Data Structure
MORE COMPLEX DATA STRUCTURE: 5 NODES, 10 LEAVES

GROUP 1
NO MODIFICATION

GROUP 2
WITH MODIFICATION

GROUP 3
WITH MODIFICATION

GROUP 4
NO MODIFICATION

GROUP 5
NO MODIFICATION

LEAF 1

LEAF 2

LEAF 3

LEAF 4

LEAF 5

LEAF 6

LEAF 7

LEAF 8

LEAF 9

LEAF 10

--- --- --- --- --- --- --- --- --- ---
POINTER GO FROM GROUPS TO LEAVES (DOWNWARD)
POINTER GO FROM LEAVES TO GROUPS (UPWARD)
POINTER GO BETWEEN ELEMENTS ON THE SAME LEVEL (TRAVERSE)

*LEAF (PICTURE DATA)
*NODE (GROUPING DATA)

Figure 11—Data Structure (cont.)
in the structure is then checked for modifications. It may be a group or a leaf. The modifications are carried out to the lower element, and then the higher group's register values replace those register values where no modification was indicated and are compounded with changed values. Compounding is done by matrix multiplying the rotation matrices, by multiplying the scale factors, and by adding translation factors (translation is additive, whereas the others are multiplicative for compounding), the answers replacing the lower element's load register values. Cutoff plane and intensity settings are simply replaced by the higher group's values. Then, bit 14 is set in the first word of the lower element to cancel any processing when the traversal algorithm reaches this element later—remember that traversal algorithm itself still points at the higher group, and will later chain through the lower elements in order to pass pointers to the display. The procedure is to this point simply compounding transformations.

Now, if the lower element is a group, it is then a subgroup to the higher group, and its address in turn is pushed on the stack, and the next lower leaf or group is consulted. The modifications, multiplications, etc. take place again, this time using the last element in the stack to retrieve the compounding information. This process continues until a leaf is reached. That leaf now has all the compounded transformations stored in its register load
instructions. It is these values only which are passed to the analog hardware of the scope. Therefore, it is only at the point that the traversal algorithm reaches this leaf and passes its address to the display, that any visible results occur. If bit 14 were not set, the traversal algorithm would recalculate the modifications based only on the leaf information and destroy the important compounded information. The traversal algorithm has not reached this point yet, however, and is still pointing at the top group. The compounding algorithm is ready for more action, though.

Assume the leaf has a sibling in the data structure. The sibling can be a leaf or another group. If it is a group, the same code re-enters, the modifications are compounded and that group's address is pushed on the stack. The level of grouping is obviously indefinite for all practical purposes. Since recursion of grouping makes absolutely no sense, it is not allowed in the sense that groups cannot point to preceding elements. Any lower elements would then be compounded on the basis of the last element whose address was pushed on the stack. If the sibling were a leaf, it is treated exactly as the first leaf was, and the pointer to its sibling is taken. Assume the next leaf is the last leaf. Its pointer points up at a group node sibling/up pointer, which is found at the fifty-second word past the group's address. That group's address is then popped off the stack and its sibling, if any
is processed using the top of the stack address to generate compounding information. If there is no sibling at this level, the pointer is an up pointer at a higher group. The compounding algorithm thus threads through the structure pushing and popping until the marker is again seen. At that point, the traversal algorithm zips down immediately to all the leaves, passing their addresses in turn to the display driver until a first word of a group or leaf is found with bit 14 cleared. Then the compounding algorithm takes over if it is a group as before, or the modifications are simply processed if it is a leaf.

It might be mentioned that the register load instructions affect the analog values of the display's hardware transformations. Rotation, scaling, translation, intensity, etc. are done by analog modifications to the vectors after they are retrieved from core. A software-only system would have to modify the vectors in core then ship the values out to the scope. Thus, the Vector General Scope is always rotating pictures—when a picture is stable, it is rotating with an identity matrix. This type of analog transformations also allows two leaves to share the same vector list, but with the controlling and register load information of each being completely different, so that the motion dynamics of each leaf can be separately controlled. (The "COFY" command sets up this linkage for the user.)
A few other things are done by the traversal algorithm after it has finished handing pointers to the driver. Each frame, that is, each time every list has been passed once, the analog devices are read, the function switches polled and the system timing table (TIM1AB) incremented. The display is then told to wait until the next clock interval to restart the process again.

As was stated in an earlier chapter, the ability to have arbitrarily complex compounded display transformations in real-time is the most visually exciting capability of this system. The fact that it is so easy for the user to specify should be more acutely appreciated at this point.

E. System Timing Table and Motion Table Interaction

The Graphics Symbiosis System has a powerful timing facility built into it which, to this point, has been only partially exploited. Every fifteenth of a second or, more exactly, four display cycles, the sixteen entries of the timing table are incremented by the traversal algorithm. If the display is overloaded, in that too much display information is being passed for shading or filling, the timing table still provides accurate timing information. Since any camera would have to be adjusted to the flicker rate of a complex picture, and the timing table is incremented at interrupt level in direct relation to the number of frames displayed, the film will later translate
timing back to real time when projected at twenty-four frames per second. Thus, six hundred ticks of the timing table represents forty seconds of film no matter how long it took to film it. Interrupt level timing is the only way to assure such accuracy.

Coupled with the timing table is the motion table (see Figure 12).
### TABLES

#### MOTION TABLE

<table>
<thead>
<tr>
<th>+± Usable Storage for Trap Bits</th>
<th>+± Usable Storage for Trap Bits</th>
<th>+± Usable Storage for Trap Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>32 Paired Entries (64 Words)</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### TIME TABLE

<table>
<thead>
<tr>
<th>± CNT or Trap Bits</th>
<th>± CNT or Pointer</th>
<th>±</th>
</tr>
</thead>
<tbody>
<tr>
<td>±</td>
<td>±</td>
<td>±</td>
</tr>
<tr>
<td>±</td>
<td>±</td>
<td>±</td>
</tr>
<tr>
<td>±</td>
<td>±</td>
<td>±</td>
</tr>
<tr>
<td>±</td>
<td>±</td>
<td>±</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>32 Paired Entries (64 Words)</strong></th>
<th>±</th>
</tr>
</thead>
<tbody>
<tr>
<td>± CNT or Trap Bits</td>
<td>±</td>
</tr>
<tr>
<td>± CNT or Pointer</td>
<td>±</td>
</tr>
<tr>
<td>±</td>
<td>±</td>
</tr>
<tr>
<td>±</td>
<td>±</td>
</tr>
<tr>
<td>±</td>
<td>±</td>
</tr>
<tr>
<td>±</td>
<td>±</td>
</tr>
<tr>
<td>±</td>
<td>±</td>
</tr>
</tbody>
</table>

#### TWO WORD PAIRS

- **FIRST WORD**
  - $< 0 \Rightarrow$ MOTION FUNCTION
  - $= 0 \Rightarrow$ UNUSED
  - $> 0 \Rightarrow$ PATH MOTION

- **SECOND WORD**
  - 16 BIT POINTER TO PATH OR FUNCTION

MOTION TABLE IS CALLED AT INTERRUPT LEVEL BY TIMEOUT OF TIME TABLE

#### TIME TABLE

- SECOND WORD IS INCREMENTED EVERY CLOCK TICK.
- WHEN FIRST WORD COUNT EQUALS SECOND WORD COUNT (TIMEOUT)
- CORRESPONDING ENTRY IN MOTION TABLE IS CONSULTED.
  - IF POINTER (NOT COUNT), SAME AS TIMEOUT

- **FIRST WORD**
  - $\geq 0 \Rightarrow$ COUNT (SET BY USER)
  - $< 0 \Rightarrow$ POINTER

- **SECOND WORD**
  - IF COUNT, 15 BIT FRAME COUNT
  - IF POINTER
    - $< 0 = $ PATH
    - $> 0 = $ FUNCTION

---

**Figure 12—Motion and Time Tables**
The timing table can deal with absolute counts, and could use counts determined by the x value on a curve in the x-y plane, and counts algorithmically generated. If a time count is expired one way or another, the corresponding entry in the motion table is consulted for the dispatch address to the routine whose execution is dependent on the timing information. Eventually, by this method, one will be able to specify any coordinate path or algorithm as a timing factor.

Presently, only path following ("PATHMOV") uses this facility. Before drawing was replaced by system utility macros, it too used the timing and motion tables for sampling frequency. The facility is obviously open for further development, when the needs arise.

As an example, path following requires that a figure be moved from point to point along a path according to the speed indicated by a dial (or some other DEV). The path following algorithm places the count from the dial setting in the first timing table word. When that count and the system incremented count match, the routine is executed and sets up the new count, and zeroes the system incremented count. Even at this point, equations for acceleration and motion dynamics of other forms may be generated within macros, using a variable for speed of path following instead of a dial. Path following is an extremely valuable tool for
animation. The Graphics Symbiosis System provides a three-dimensional tangential path following capability, examples of which can be seen in Figures 13 and 14.
PATHMOV— A swallow flapping its wings and moving about a path in real-time.

Figure 13—Pathmov Example I
Figure 14—Pathmov Example II

PATHMOV—Any object can be moved tangentially along any 3-D path with Z-axis motion simulated by proportional reduction in size of the object as it moves "away" from the user.
F. Grouping and Data Structure Manipulation

Essentially two data structures are maintained in the system. The displayed data structure is the one used by the traversal algorithm. The non-displayed data-structure is used for holding pictures that are selectively erased but will be used again. Both structures are threaded trees which were modified slightly. Elements can be removed from groupings by passing them from one structure to the other by the "PUTLIB" and "GETLIB" commands. Links are of the one-way type, since the extra effort necessary for deleting items from one-way linked lists due to searching is not significant in this case. Standard methods are used for pointer chaining, except some timing waits were put in to prevent a list that is currently being displayed from having its pointers changed.

As in allocation and deletion, grouping is not a particularly time-sensitive operation. Whereas extreme care was taken to make the interrupt level code as efficient as possible because it is directly related to the number of vectors that can be displayed in real-time, allocation, deletion and grouping run at normal level and any inefficiencies due to searching and sorting do not result in more flicker, or for all practical purposes, the system being any less interactive. In the unlikely case that a user would spend all his time allocating, deleting and
grouping, and nothing else, he might be slightly penalized as a result of the methods used for these functions.

G. Names Building and Accessing

The Graphics Symbiosis System was designed to make the use of names as convenient as possible. The system resident names table includes all the system functions, followed by a macro parameter passing area, followed by the user names, all in ASCII format (see Figure 15).
### NAMES TABLE

<table>
<thead>
<tr>
<th>Name 1</th>
<th>System names are first, followed by user names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name 2</td>
<td>User names are 1) Pictures 2) Disk resident functions</td>
</tr>
<tr>
<td>Name 3</td>
<td>System names are consulted only when syntax calls for a system name so, &quot;DRAW DRAW&quot; where the first is the keyword and the second is the name of the picture is possible</td>
</tr>
</tbody>
</table>

This scheme allows quick access to names.

### NAMES DISPATCH TABLE

- Pointers for name 1
- For name 2
- For name 3
- Etc.

*Figure 15—Names Table*
Access to the table is from any of these three entry points, depending on how the name is to be used. For instance, "ROTATE ROTATE,X,DO" is permissible if the user has a picture called "ROTATE." All functions that could deal with the picture "ROTATE" access names from the beginning of the user names table, rather than from the beginning of the system names table where the function "ROTATE" is found. Any confusion is thus avoided. To refer to a name, only enough letters to uniquely identify it need be typed (except for disk access functions for which the whole name must be typed). Thus, "R R,X,DO" is sufficient if "ROTATE" the function and "RCTATE" the picture are the first occurrences of "R" in their respective names tables. If the picture name were "ROBIN," "R R,X,DO" would still work, of course, providing "ROBIN" were the first name beginning with "R" in the user table.

The system will not let the user create a name that could cause a multiply-defined name conflict. Thus, if a name like "TEST3" exists, the user cannot have another picture or macro named any of "TEST," "TE," "TEST3," "TES," or "T" until "TEST3" is deleted. An error message informs the user if he tries to multiply define a name.

The names table access routines also have a "wild card" capability. For example, if "DELETE TE*" were typed, all names beginning with "TE" would be deleted. At this point,
however, none of the system routines yet acknowledge this capability, a state of affairs which should improve soon. Note, though, that an unnamed macro, namely:

```
<DELETE TE
  SKIP -1>
```

would do the same thing, stopping when all names beginning with "TE" were deleted by printing out an error message.

The system was designed to leave room for expansion and thus, many "hooks" are left in the software. An example of this is the presently unused eighth character that every name has in the names table. This character is never passed, but may be used for flags, if that need arises in the future. Every name is also set up to pass switches, although only the system function names are presently consulted for switch values.

One further names-oriented routine is the "RENAME" command which is often useful for resolving the name conflicts in which the use of abbreviated names results from time to time. "RENAME" simply replaces the old name with the new name letter for letter in the names table.

Clearly, the names table routines are not milestones of computer science. The explanations are only included to round out the reader's knowledge of the system.
H. Critical Comparison of this System with Three Other Systems

In the Introduction, several computer animation systems were briefly examined. In this section, three of these systems—CAFE [4,28], GENESYS [14,29] and the National Research Council of Canada System [9]—will be contrasted and compared to the Graphics Symbiosis System in enough detail to document the differences in language power, quality of interaction, and the relative level of habitability and extensibility in each of the systems.

The Graphics Symbiosis System was designed for habitability and extensibility. Since these terms and the concepts they represent are fairly new, most likely these other systems did not explicitly have habitability and extensibility as goals. GENESYS, of course, was designed to be very habitable, but the other two systems seem to be oriented more towards film production than towards advancing the state of the art in programming languages. The application of these design criteria to the Graphics Symbiosis System has resulted in a product that is more powerful, more interactive, and more of a true programming language than previous efforts in graphics, only part of which is due to better hardware. As hardware becomes more complex, it gets increasingly challenging to make good use of the augmented power.
Before further discussion, the concept of a programming language must be defined. A suitable definition in the present context is a command set that includes some form of subroutine calls and looping facility. Therefore, the capability of concatenating command strings by itself is not a sufficient reason for calling a command set a programming language. Similarly, an interpretive effort such as the typical interactive text editor cannot be called a programming language (an exception is PDP-10 TECC which can have named macros and loops). This definition of programming languages may be too strict but it represents an important distinction, especially in interpretive languages.

CAFE is not an interactive system in the sense of the discussion here. Commands may be prepared interactively in the same manner one creates source language programs on a time-sharing computer terminal with an editor, but they must be further processed before displaying. There is no run-time interaction. The 2250 display, in fact, is used for viewing the picture sequences prior to microfilming for proofing purposes—an excellent approach for a microfilm-oriented system.

CAFE uses the admirable approach of combining SNOBOL4 (for language transformation and data referencing), and FORThAN (for numerical aspects). SNOBOL, of course, has very powerful string manipulation and data structuring
capabilities, two essential features for using a higher level language when implementing an animation graphics system. Unfortunately, CAFE's language is not a programming language. There are no program flow control statements, no conditionals, no subroutines, and no numeric or alphanumeric variables. Secondly, CAFE is not interactive in that all picture sequences are determined by previously prepared command files. The SNOEOL part of the implementation makes this command language quite attractive otherwise.

Every point in a CAFE picture has a name. Points make up objects which also have names. Objects combined with motion instructions make up "graphs." Since there is no interactive drawing facility in CAFE, it is fortunate that the authors devised a good method for describing points. In most systems, description of a unit cube takes forty-five coordinates (fifteen points) because a cube cannot be drawn with a continuous line without backtracking. In CAFE, the eight corner points are named and the interconnecting lines are specified in one statement. A slash indicates a drawn vector, and an asterisk indicates a blanked vector. Thus, the twelve visible lines in the unit cube are represented as follows: "a(0,0,0) / b(0,0,1) / d(0,1,1) / c (0,1,0) / g(1,1,0) / h(1,1,1) / f(1,0,1) / e(1,0,0) / a / c * g / e * f / b * d / h." This approach is particularly useful when many lines radiate from several of the vertices.
Given this data specification, one can easily refer to points within objects for purposes of generating data. In contrast, the Graphics Symbiosis System allows the user to interactively create and identify displayed points with the lightpen, dials, or other devices so that each point in a picture does not need a name. Then, these points taken as numbers may then be used for whatever purpose the user desires—for creating new data, setting scale, translation, rotation or intensity values, or as inputs to numerical calculations within macros.

The transformations in CAFE are done by software and necessitate point-by-point calculations. The original picture description as named points is never modified, however, so the transformations are always erasable. Rotation is not very flexible in this system since only simple rotation around coordinate axes is provided. Complex, arbitrary-axis, arbitrary-origin compounded rotation is necessary to display swimming fish and birds with flapping wings, and this type of rotation is not possible in the CAFE system. Furthermore, all of CAFE's objects are stick figures—no shading commands exist.

If CAFE were a truly interactive system with real-time response, the system's data specification and object description would constitute a fairly powerful system. The concept of "scenes," that is, sequences of picture modifying
commands, could easily be carried over to an interactive approach, as could the user-oriented command syntax. The CAFE system also lacks a path following facility, a technique which was shown to be essential to animation graphics by the next system to be discussed.

GENESYS is a highly interactive system which exploits the concept of two-dimensional path following to achieve animation sequences. The user draws his pictures on a data tablet and the image is recorded in real-time. Since the drawing is displayed, the user can watch the screen while drawing. The same device is used to move a cursor which, with a button push, allows the user to select modes of operation from a displayed menu. The path following curves are generated in the same way as pictures and the system relies heavily on the fact that by sampling at a constant rate, more points are recorded per unit distance the slower the user moves the tablet pen. If he moves the pen rapidly, fewer points per unit distance are recorded. When the image is put on the path, if the time interval between points is again constant, the path following controls time relationships as well as x-y plane positioning in the exact way that the user specified the motion. Thus, the user can control his pictures in the same way he controls the position of his hand on a two-dimensional surface.

In conventional animation, motion is simulated by
changing the position of objects on an animation stand. Computer animation is essentially the act of replacing the animation stand. To change the shape of an object in conventional animation, a new picture must be drawn and inserted in front of the camera at the proper time. If the object continually changes shape, up to 14,400 drawings might be required for a ten-minute animation sequence. GENESIS automates this changing of object procedure in an innovative manner. The user can substitute one picture for another at any given time. The substitution procedure involves interacting with a displayed row of dots each representing a time slice of a fraction of a second. By moving the cursor, the user can specify which of the time slice dots are to be assigned to the substitute picture. By repeating this procedure, a varied animation sequence can be built up. Clearly, to generate a thousand frames (about forty-two seconds), a lot of dot manipulation could be involved if much image metamorphosis were involved.

GENESIS also allows the user to specify timing by tapping a function switch, thus adding another means of tactile interaction. Further features of the system permit editing of motion sequences and some positioning of objects with dials.

Ron Baecker himself describes the hardware he used as "stone-age" [15, p.265]. His work is admirable in that some
of the philosophy of interaction in animation graphics was explored to a great depth. GENESIS, however, has serious shortcomings. The picture quality is poor and the implementation is totally two-dimensional. The command language is minimal and is not a programming language in the sense described above. Of course, there is no macro facility. Shaded images are not possible and the lack of three-dimensional representations preclude all but z-axis rotation and the rotations that can be simulated by one-dimensional scaling.

Furthermore, the picture substitution approach to computer animation is not conducive to smooth transitions between pictures. To achieve this kind of transition, one would have to specify a change of picture at nearly every step, and draw all the in-between pictures. This procedure cannot be automated easily, and is not nearly as effective and exciting as the totally automated linear interpolation (blending) as it exists in the National Research Council's system or the Graphics Symbiosis System. Furthermore, the path following in GENESIS is not even tangential. The Graphics Symbiosis System allows three-dimensional tangential or non-tangential path following (user option) with speed controlled by the point density of the path, optionally compounded with dial settings or interactively specified algorithms. A form of picture substitution similar to that of GENESIS can be done with macros.
Furthermore, a blending picture can be moved along a path, rotated, the path and the picture rotated, translated, scaled and combined with fade-ins and fade-outs. The path can be put on a path, as in the case of ejecting the contents of an airplane cockpit along a path while the airplane is flying along another path, and this process can be repeated to any desired level. The picture quality of the Graphics Symbiosis System is far superior to that of GENESYS. In other words, one is a production system and the other is not. GENESYS does represent the first habitable animation system and its implementation introduced many new concepts. Both the GENESYS film [30] and Csuri's film "Real-Time Film Animation" [31] can be acknowledged as influences on the design of the Graphics Symbiosis System.

Another system designed for production and really used for production is the National Research Council system. The system is oriented towards the professional animator and is based on conventional cel (a transparent overlay) animation techniques. During the design stage, the animator manipulates wire-frame figures which can later be converted to raster scanned shaded pictures for filming. Burtnyk and Wein are to be credited with the first useful implementation of linear interpolation [9].

The N.R.C. system uses a combination of the "menu" technique and a somewhat cryptic command language whose
operands are mostly numerical. The menu is used for the interactive part of the system which includes two-dimensional drawing and picture modification routines ("GRABBER," "PULLER" and "SHAPER"). All the motion sequences are prepared off-line using the command language so there is no interaction during run-time. In fact, Burtnyk and Wein maintain: "Neither the use of a drawn path nor the generation of a path by continuous interactive control have been found satisfactory except for purposes of experimentation." The users of the Graphics Symbiosis System would strongly disagree with this conclusion.

Within the command language it is possible to specify frame numbers and indicate the interactions between any of several levels of "cels" during any particular frame. Several different types of transformation may take place simultaneously on one or more pictures, if so indicated in the command string. One must specify the type of velocity curve (decelerating, accelerating, linear, etc.) mapping the rate of change for each transformation applied to the picture. All parameters are specified numerically and very exacting motion sequences may be generated.

The linear interpolation will preserve boundaries if some conventions are preserved. Thus, shading blending images is possible, resulting in merging solids, a very striking visual effect.
The command language of the system is not a programming language in that it is only a long string of sequentially listed commands which control several simultaneous operations. Any new additions to the language would have to be done in assembler.

The Graphics Symbiosis System allows control of any picture or hierarchy of pictures with motion sequences algorithmically specified or interactively generated. At least forty or fifty individual pictures can be manipulated separately and up to one hundred square inches of shading can be displayed in real-time (due to a high-speed vector generator), all while moving, path following, rotating, and flapping wings. Recent implementation of a boundary preserving linear interpolation technique provides the capability of blending solids as is done by Burtnyk and Wein. The Graphics Symbiosis System is much easier to use as well, since parameters rarely have to be specified as numbers, and the language is fairly easy to learn and read. The program flow control statements and macro facility allow the user to define acceleration and deceleration in terms of paths, algorithms, dials and switches. The quality and quantity of the interaction is much greater than with the N.R.C. system.

The N.R.C. system has no facility for compounding transformations, so all complex motions must be specified in
terms of linear interpolations. This may, in fact, turn out
to be a better approach to automating complex walking motion
animation and other such phenomena. Of course, it's best to
let the user decide, since linear interpolation may not be
usable for the generation of scientific-type films which one
is likely to desire in an educational environment.

The Graphics Symbiosis System owes its flexibility and
productivity to the fact that it was designed for
extensibility and habitability, and uses state-of-the-art
equipment. As the electronics become more sophisticated and
new modes of thinking are adopted, it is hoped that this
system, if nothing else, can be used to demonstrate that a
powerful, easy-to-use interactive system is worth the effort
of implementation.
Chapter IV—The Future

The discussion so far has covered the features of the Graphics Symbiosis System that are implemented and working. This chapter will consider new features which will enrich the system and will undoubtedly keep the system programmers busy for a while. The extensions are not primarily to the power of the programming language, but are more generally oriented toward making the animator's tasks even easier. The topics presented are more or less in the order in which they will be implemented, the last couple being somewhat fuzzy at this point. A few words will be said at the end of the chapter about work that needs to be done in habitability and extensibility studies.

First, a "TRACE" command will be written. "TRACE" will record the states of the analog devices and the function switches, as well as any keyboard input from the moment the user types "TRACE" and will continue until disabled. Only changes of values will be recorded every fifteenth of a second, so that even in the case of the user turning half the dials continuously, a good half-hour could fit on half a disk cartridge (600,000 words). A half-hour represents a lot of film. Note that the variables do not have to be recorded since they are algorithmically generated or read from dials. The time represented in the file will also be real-time, since it will result in the same amount of film,
regardless of the time it took to generate complex pictures. If the user creates a half-hour long "recording," he should have accurately reproducible sequences to the point that he could conceivably set up the camera with a thousand feet of film and go have dinner. "TRACE" will go a long way towards automating the filming, and thus would increase the system's habitability.

Linear interpolation is a technique whereby the animator specifies two pictures and the interpolation routine generates the "in between" frames to produce a smooth transition from the first to the second picture. Thus, most of the drudgery of blending images is automated and a new tool is given the animator. Peter Foldes, in conjunction with N. Burtynky and W. Wein of the National Research Council of Canada, has produced films [10,27] in which motion is approximated by successive interpolation sequences, which is essentially how conventional cartooning is done. The technique needs to be exploited in a wider field of application than Foldes' admirable work is based, and the Graphics Symbiosis System is the ideal vehicle for this work because blending pictures can be rotated, scaled, moved along paths, and generally treated like regular pictures. The linear interpolation technique presently available to the system preserves boundaries which permits shading to occur at each intermediate frame (at somewhat less than real-time, averaging about two to four seconds per
frame), thus creating sequences of blending half-tone solids. Another improvement was to allow parts of the first picture to selectively blend into parts of the second picture, thereby eliminating the current blending of heads into feet and other such normally undesirable effects. Linear interpolation is a technique which does much to automate the production of quality film, thereby enhancing the system's habitability.

Concurrently developed will be data generation facilities, the most sorely needed aids to computer animation at this point. The work has been started with "WARF" which allows flexible manipulation of boundaries (or any vector list), and "FILL" which permits the user to fill boundaries with randomly placed "fill-characters" of his choosing, thereby creating textures (see Figure 16).
Figure 16—"FILL" Example

An example of the "FILL" routine. The "fill character" in this case is a small straight line which is randomly placed and randomly rotated within the specified outline.
The sonic pen will be developed as a three-dimensional drawing tool, perhaps through the use of macros, possibly through off-line programs with different data structures. A prototype modeling macro which allows the building of complex pictures from user-created simpler elements was written in a single morning. This macro represents a level of flexibility and sophistication which would take months to achieve on a new machine with little or no graphics software available. Various curve fitting and parabola generating techniques will be exploited to help artists and instructors describe their conceptions to the computer. Perhaps generated perspective views on a split-screen or two monitors with adjusted parallax will be used to aid in viewing three-dimensional structures being built. The data generation problem is a user-oriented one in that each step in solving it will increase the ease of producing images, a definite improvement in the level of habitability.

Then, a full FORTRAN-like arithmetic capability will be added, complete with floating point variables, precedence operators and nesting. Since the interpretive approach is somewhat lacking for complex numerical calculations, it is planned to compile arithmetic statement groups into PDP-11 assembler code, create a swap module for the code during disk-to-core transfer, and use the module within a macro like any other assembler swap module. Thus, very efficient numerical processing will be possible. Subroutine calls to
assembler routines for sine, cosine, matrix multiplication, and others also will be included, and thereby increase the capability of the language for numerical computation. Adding such capabilities is a conventional problem in computer science that has been solved many times. The implementation is simply tedious.

The system presently contains a debugging technique which was patterned after the standard Digital Equipment Corporation's "DDT" or "CDT." This debugging technique has been extremely useful to the systems programmers for aid in quickly isolating and correcting errors in the code. What is still needed is a macro-level debugging technique, a feature which will add to the system's habitability since complex macros will then be easier to construct and make usable. Presently, the "LIST" command helps debug macros by listing statements on the display terminal as they are executed. The user may interrupt a program and query a value by using the "PROMPT" command. A more advanced technique will be written once enough experience has been obtained in writing macros to discover where help is needed. Present programming languages tend to be inadequate in terms of on-line debugging aids, so much of the work would have to be original. Better macro editing facilities will be needed, but this is again a question of tedious implementation.
The idea of introducing color is a very exciting one. Of the many techniques available, none is ideal in all respects. Color cathode ray tubes exist, but the color is fairly "washed out," cannot be blended effectively, and the whole spectrum is not represented. External processes like DATACOLOR are good for translating grey levels to colors assuming the raster is fine enough for projection quality and fringing does not occur.

Recently, the support for a color wheel was added to the Graphics Symbiosis System. The color wheel (see Figure 17) revolves at 1,800 r.p.m. and is synchronized to the line frequency (60 c.p.s.).
COLOR WHEEL — The color wheel is a rotary four color disk which with proper display synchronization results in full color real-time images when viewed or filmed. Colors may be dynamically blended and assigned any of several intensity levels.
The software amounted to about eleven lines of new code, and includes a new command "COLOR." The hardware came to less than one hundred dollars. The color is easily specified, brilliant (except for red, which is a little washed out due to the spectral deficiencies of the F-4 phosphor), and can be readily filmed. Color may be specified by having the axes of the sonic pen keyed to the three primary colors, with motion of the pen resulting in spectral variations. The color wheel approach is limited, however, because the time allotted each color is less than 1/120 second, and thus the number of vectors in colored shaded pictures that can be displayed is much less than can be put up flicker-free without the color wheel. This problem will be partially solved when the color wheel is interfaced to the Arriflex camera through the PDP-11, with pulses controlling the shutter of the camera and the stepping motors of both the camera and the color wheel. Color for complex pictures will then be somewhat less than real-time, but the film will be of high quality.

Presently there are commands in the system to aid filming ("FILM" and "UNFILM"). These commands will be modified and added to when the interface of the Arriflex camera to the PDP-11 is completed.

Another project to be started is the use of the segmentation hardware of the PDP-11/45. This mini-computer
allows addressing to 128k words with sixteen bit addresses by the hardware memory management unit. The first task will be to get at the 16k of core already available, but presently inaccessible. The code has been written for this task, but it is not yet interfaced to the system. The segmentation hardware may eventually be used to exploit the Graphics Symbiosis System's inherent re-entrancy. The entire system keeps its storage in a separate section of core so that multiple users could be permitted by duplicating only the storage area for each user. Only one copy of the system code would be needed. Obviously, a multi-user system would require more than one monitor. Fortunately, the Vector General hardware can handle up to four scopes, each with its own analog input devices, function switches and light pen, at the cost of sharing the vector generator.

Much work still needs to be done in computer graphics that has not even been attempted by the implementation of the Graphics Symbiosis System. Many problems are simply too complex to handle in real-time. For example, the display normally shows all sides of a three-dimensional object, both front and back, both outside and inside. Removing the lines and surfaces of objects so that the front surfaces would block out the back, as is the case with opaque objects in real life is a problem whose solution tends to use inordinate amounts of computer time, getting worse as the
picture complexity mounts. Faster general purpose and special purpose hardware may eventually allow hidden line and surface calculations for the concave case to be undertaken in real-time. The Graphics Symbiosis System has the capability of removing hidden surfaces for totally convex objects in real-time, but only a small class of objects are convex from every view (see Figure 18).
A) A shaded 3-D "steeple". Every other surface component has been deintensified. B) The same steeple, but with the hidden surfaces removed.

Figure 18—Hidden Surface Example
In any case, with new hardware or without, new algorithms will have to be worked out for use with pictures of useful complexity. The recent work at the University of Utah by Watkins [39], Gouraud [42] and Parke [20] with shading and softening of the shaded surface edges is, hopefully, an indication of what can be done.

It must be emphasized again that the most difficult unsolved problem in computer graphics is still that of data generation. One approach is using shaded origami-like objects (see Figures 3, 4, 13, 14, 19).
Figure 19—"Origami" Fish
A) The outline of a fish composed of planar surface outlines
B) The shaded fish in 3D.
The thrust of future graphics efforts should be in this direction in order to make the description of three-dimensional input to the computer possible in much more habitable ways than presently exist.

Work needs to be done in the general fields of habitability and extensibility studies, as well. Standing in the way of increasing habitability are many obstacles, some of which can be overcome by faster or special purpose hardware, some by software. Software may be developed to parse natural language input, although linguists today know the problem is significantly more difficult than it was thought to be ten years ago. More hardware along the lines of analog input devices is clearly feasible (e.g. Noll's tactile communication work [33] and Sutherland's head-mounted display [40]. New approaches ranging all the way to electro-encephalographic techniques (reading alpha waves) could conceivably improve communication, leading to a level of direct interchange at near electronic speeds. The Graphics Symbiosis System has made innovative use of the hardware available to it and should be a good vehicle for the exploration of new hardware techniques for improving habitability and for real-time solution of difficult problems. Experiments may be done in which habitability is measured statistically in controlled environments, but the area of study then becomes one of man-machine psychology, and is thus outside the scope of the present discussion.
Extensibility in interpretive languages is likely to become more popular since, in general, computer time is becoming cheaper and programmers more expensive. Interpretive languages help people communicate their problems to computer systems and extensible interpreters allow the rapid generation of special purpose systems that themselves can grow with needs. The methods used to process syntax may be studied and progress can be made in the area of compiling interpretive segments that have been checked out to satisfaction and are ready for production tasks, following along the lines of Schuman's work on compilable interpretive extensible languages discussed in Chapter II [18].

The best method of finding out areas of needed improvement in extensible languages is to implement them, as has been done here in a very specialized situation. T.E. Cheatham is in agreement with this philosophy:

There are clearly many who feel...that we have a long way to go in basic research and development of extensible languages and their host facilities; I do not share this opinion. Rather, I think that in a very strong sense "we have arrived"—that there exist languages and host systems which fulfill the goals of extensibility.

This is not to suggest that the problem is completely solved.... However, it is my position that further research in extensible language, per se, is probably destined to yield only marginal results until such a time as significant period of exploitation has clearly pinpointed issues with which a new wave of extensible language research might attempt to
Beyond more implementation of extensible languages, Cheatham suggests that the process of programming be automated—a logical step from the very high level of coding possible with extensible languages. For this, theorem proving and correctness of program techniques will have to be refined, and the area of study quickly involves artificial intelligence, pattern recognition and mathematical logic.

Computer graphics is an excellent area in which to pursue the implementation of languages which are more extensible and more habitable, for the reasons that the users are typically more or less naive but their animation problems can be as complex to solve by computer as those of any scientific endeavor. The future of computers as tools for use by the general public lies in providing computer systems which are easy to learn and use—ones which the user, with brief instructions, can describe his problems and have computer systems quickly adapted to his needs.

The people who have worked on the Graphics Symbiosis System in the past year are confident that the state of the art in computer graphics has been significantly advanced by this project.

There are two essential reasons for the present paucity of computer graphics installations and the relative inaccessibility of existing ones. The first is cost. Heretofore, the only hardware available of comparable power for animation graphics involved a PDP-10/Evans and Sutherland LDS-I system that runs a minimum of $600,000. The PDP-11/45 and Vector General configuration at The Ohio State University represents an investment of approximately $160,000. A minimal PDP-11/20 system with a 3-D scope would cost about $80,000 and have everything but half-tone shading (which requires floating point calculations in the present implementation). A two-dimensional scope is still less expensive (one could still do rotations by software). Admittedly, a PDP-10/LDS-I system represents more powerful hardware, but not in the order of four to six times as powerful. An IBM 1130/2250 system costs twice as much as the CSU system above, but it is at least twenty times slower and has no rotation or floating point hardware.
Thus, it may be observed that by simply diverting funds for a high-speed line printer from the local computer center (maybe a tape drive or two also), a decent stand-alone graphics system may be purchased. The cost of such systems will decrease as soon as volume sales justify the abandonment of hand-wired modules. Even now, the equipment cost is within the reach of many teaching aids laboratories, state institutions and computer assisted instruction centers.

It costs more today to develop software than to purchase hardware. For this reason, the Graphics Symbiosis System has been carefully modularized to allow its use on systems of less complexity than the CSU system. Feature switch settings permit or prevent assembly of floating point code, hardware or software multiply and divide instructions, and polling and accessing dials, function switches, and VT05-like devices, in any combination. Features like the keyboard substring matching facility or the login and password restrictions may be conditionally assembled. All the feature switches are in a system parameter file which may be edited to fit a particular configuration. The intention is to eventually distribute the software.

The second reason that has prevented general accessibility to computer graphics installations has been the high level of expertise normally needed to program
packages for computer graphics applications. In that it is a general purpose graphics programming language, the Graphics Symbiosis System has eliminated, to a great extent, the need for the expensive coding of what have been called "throw-aways" or one-shot, single purpose programs. Furthermore, the habitability considerations included in the design of the language allow a non-programmer who could not effectively program in FORTRAN to interact with the system to make useful films in relatively short periods. The first film made on the system, using unsophisticated newsreel camera equipment, lasted twenty-five minutes and was shot entirely in the week before it was shown for the first time. About eight hours of setting up and filming were required, and the level of expertise was somewhat lacking at that point. The images were generated beforehand, but the sequences were all developed as shooting progressed. Such an effort using conventional techniques could easily have taken several months.

In recent weeks, the Graphics Symbiosis System has been put into production use at The Ohio State University. The first naive users are now writing macros and making film. New users express astonishment at how easily they can manipulate pictures. They had all read the command descriptions and some introductory material and seen examples of film made on the system but claimed that the verbal and visual documentation did not do justice to the
part of the system that was most important to them—what we have been calling "habitability." It is hard for the beginner to appreciate the quality of this essentially tactile and visual interaction by reading about it. Sophisticated computer graphics audiences have been able to appreciate the nature of this system, but even then, film was necessary to support the discussion. Computer graphics gives what is essentially four-dimensional output: the two-dimensional movies give the illusion of three dimensions through motion and the fourth dimension is conveyed by the time the film takes. To explain this system in terms of one-dimensional English syntax, or two-dimensional photographs is a task that rarely succeeds, especially when aimed at the uninitiated.

To document the realism that can be achieved within the system, an incident that happened during the final editing stages of the first film can be related. The film was being shown in the presence of several people and a neighborhood stray cat who tends to be quite lethargic. It has been remarked that nothing save physical violence and petting gets a reaction from this cat. During the first several minutes of the film, the cat was watching the screen, half-asleep. When the shaded, three-dimensional "origami" swallow (see Figure 14) with its sinusoidally flapping wings came on, the cat literally went wild and climbed the screen. After the swallow sequence, it left the room. In a day and
age in which television sets are advertised as being superior because live cats paw at images of parakeets in bird cages on the screen, this incident must mean something. The image quality is by no means good enough to fool humans, however.

Video tape systems could be coupled with computer graphics equipment for general purpose special effects generation. Indeed, on-line, real-time graphics with video backgrounds or foregrounds is a real possibility for commercial and educational television environments. Using video disk techniques, and a system like the Graphics Symbiosis System, colors may be introduced and automatic masking of backgrounds used to create high quality video output for entertainment and educational use.

Computer assisted instruction efforts have been hampered by the high cost of graphics, so that only a few systems use visual aids more complex than a slide projector. The Graphics Symbiosis System could be used to prepare images for use on either a storage-tube display terminal (which looks like a teletype to a computer's operating system) or a small, low-cost vector generating display terminal of the type recently announced by both Vector General and Digital Equipment Corporation. These terminals represent a viable alternative to fifteen character-per-second text oriented computer assisted
instruction systems, especially when fast hardcopy facilities are added.

This project's true short term benefits to society will probably be realized in the inexpensive generation of film or video tape. This film or tape will communicate information that has been too expensive or too time consuming to conventionally animate, or that is not really possible to explore otherwise, as in some topics of the mathematical and physical sciences. Recently, a video tape was generated for the Mathematics department illustrating some concepts of geometry and two multi-media shows used many minutes of computer graphics to illustrate shifting continents, evolution of lystrosauri, the Fibonacci series in plants, the three-dimensional nature of complex molecules, among other things. These efforts were accomplished rather rapidly, the mathematics film in about two hours.

Mixed media of this type can facilitate the introduction of sophisticated concepts to students, or even to the general public. One project might be to illustrate birth control concepts graphically to those persons least likely to understand overpopulation and the need for zero population growth, that is, the persons most necessary to reach. Another sequence could simulate a rail system on existing tracks that could alleviate commuter auto traffic
in cities without subway systems. Productions like these could be distributed at low cost on video tapes to be shown in neighborhoods. The potential of computer graphics is clearly unbounded.

The possibility of some elements in society using such a tool to indoctrinate is unfortunately present. In fact, much of the animosity one perceives towards computers in general is due to the wide misuse of information storage and retrieval facilities. One has to sometimes rationalize not being a modern-day Luddite in this respect, and hope that the legal apparatus of this country, and other countries, will obstruct misuse of computer graphics, computer assisted instruction, and data banks as vigorously as it has other undesirable elements.
REFERENCES


[23] DATACOLOR Colorizer, Spatial Data Corporation, Goleta, California.


[38] A. M. Noll, Real-time Interactive Stereoscopy, Bell Labs, no date.


Appendix A. Language Command Syntax

This Appendix gives the syntax of each command of the system. The printout is from the "HELP" message file. The commands are listed alphabetically (with some very new ones added on the end) and are printed exactly as they would appear on the display terminal. This appendix was produced by changing the command device to the hardcopy terminal, an option available to any user.
HELP

SYNTAX: HELP COMMAND-NAME

HELP IS HERE TO ANSWER SYNTAX QUESTIONS.
CERTAIN ABBREVIATIONS HAVE BEEN USED WHICH WILL, HOPEFULLY, AID UNDERSTANDING:

NAME  IS ANY PICTURE OR MACRO NAME
FRAME  IS ANY PICTURE NAME (GROUP OR SINGLE PICTURE)
LNAME  IS ANY SINGLE PICTURE (USUALLY CALLED A 'LEAF')
GNAME  IS ANY GROUP NAME
MNAME  IS ANY MACRO NAME
DNAME  IS ANY DISK FILE NAME

NAMES MAY BE ABBREVIATED TO ENOUGH LETTERS TO INSURE UNIQUENESS.

DEV  IS ONE OF THE FOLLOWING ANALOG DEVICES OR VARIABLES
  DIALS 0-9      (D0-D9)
  JOYSTICK      (JS OR JX, JY, JZ)
  SONIC PEN     (SP OR SX, SY, SZ)
  VARIABLES A-Z (A-2)

EXPR  IS A MIX OF NUMBERS, DEV'S AND ARITHMETIC OPERATORS WHICH ALWAYS EVALUATES TO A SINGLE NUMBER.
THE OPERATORS ARE +, -, *, /.
EXAMPLES:
A
D7
200
A+B/200+35-B*17

EXPRESSIONS ARE CURRENTLY EVALUATED FROM LEFT TO RIGHT
WITH NO OPERATOR PRECEDENCE SO THAT 20+2/2 IS 11 NOT 21.

ALL COMMANDS EXCEPT ON, IF, CALL, AND DO ARE DOCUMENTED. THE DESCRIPTIONS MAY BE CONSULTED BY TYPING HELP FOLLOWED BY THE COMMAND NAME. MACRO SYNTAX ALSO IS NOT DOCUMENTED AT THIS TIME.

REMEMBER TO USE THE PANIC BUTTON IF THINGS GET OUT OF HAND, AND THAT THE WORST YOU CAN DO IS STALL THE COMPUTER.
HELP RESET

SYNTAX:  RESET NAME

RESET IS THE SAME AS FIX EXCEPT THAT THE ORIGINAL VALUES OF THE
ROTATION MATRIX, ORIGIN, SCALE, INTENSITY, ETC. ARE RESTORED.

SWITCHES:
SAME AS FIX

EXAMPLE:  RESET/R PROP

HELP RETURN

SYNTAX:  RETURN

RETURN POPS UP ONE LEVEL IN A MACRO BEING EXECUTED AND THUS
PROVIDES CONVENIENT EXIT TO THE END OF THE MACRO.

EXAMPLE:  TOH: <Y=0
          SAI: <Y=Y+1
          IF Y GT 10, EXIT
          IF Y GT 5, Proc Y+Y
          IF Y GT 1, RETURN
          IF Y LT 10, GOTO SAI
          FROM "THOSE HERE THE SQUARES OF 5 THROUGH 10"
HELP GETHIT

SYNTAX: GETHIT N,X,Y,Z,K


EXAMPLE: GETHIT F,U,V,W,G

HELP GETLIB

SYNTAX: GETLIB PNAME1, PNAME2

GETS PNAME1 FROM NON-DISPLAYED DATA STRUCTURE (I.E. PUTLIB'D) AND PUTS IT INTO THE DISPLAYED DATA STRUCTURE AFTER PNAME2. IF NO PNAME2, THEN PNAME1 BECOMES THE FIRST ELEMENT IN THE DATA STRUCTURE. PNAME1 MUST HAVE BEEN PUTLIB'D OR GOTTEN IN .BIN FORMAT IN GETDSK. PNAME2 MUST BE DISPLAYED.

EXAMPLES: GETLIB SAM  
            GETLIB MARY, SAM

HELP GETPOI

SYNTAX: GETPOIN LNAME, N, X, Y, Z, K

GETPOIN GETS A POINT FROM AN LNAME. THE X, Y, Z COORDINATES OF THE POINT INDICATED BY THE EXPRESSION N ARE RETURNED IN VARIABLES INDICATED HERE BY X, Y, Z. N RANGES FROM 1 (THE FIRST POINT) TO THE LAST VECTOR IN THE LNAME (DEPENDS ON HOW BIG THE PICTURE IS). K IS A VARIABLE IN WHICH THE FOLLOWING IS INDICATED:

K=0  DRAWM VECTOR
K=1  NON-DRAWN VECTOR (JUMP)
K=-1  END OF LIST (LAST VECTOR)

EXAMPLE: GETPOIN MITCH, 20, G, H, I, K  
THIS WILL GET THE COORDINATES IN DECIMAL OF THE TENTH VECTOR IN MITCH AND PLACE THEM IN VARIABLES G, H, I AND INDICATE WHETHER THIS LINE WAS DRAWN, A JUMP OR THE END OF THE LIST, IN VARIABLE K.

HELP GOTO

SYNTAX: GOTO MNAME+EXPRESSION

GOTO IS USED TO TRANSFER CONTROL TO NAMED MACROS EITHER INTERNAL OR EXTERNAL TO THE MACRO BEING EXECUTED.

EXAMPLE: TOM:<X=0  
          SAM:<X=X+12  
          IF FS0=0, GOTO SAM

THIS EXAMPLE WILL INCREMENT VARIABLE X UNTIL FS0 IS PUSHED
HELP FSON

SYNTAX: FSON EXPR1, EXPR2, EXPR3...

FSON turns on (sets to one) function switches corresponding to EXPR1, EXPR2, etc. FSON does not work with FS15 or the panic button.

EXAMPLE: FSON 0, 2, 4, 6, 8, 10, 12, 14

This turns on the even function switches so that they light up.

HELP GETCOM

SYNTAX: GETCOM DNAME. EXT

GETCOM is the same as GETDSK except it takes DNAME from the common area on the disk ([30, 1]).

GETCOM is equivalent to:
GETDSK DNAME. EXT([30, 1])

HELP GETDSK

SYNTAX: GETDSK DNAME. EXT([30, X])

GETDSK gets the DNAME from the disk area indicated by "[30, X]" (default is your area). If no .EXT, it defaults to .DEC. DNAME becomes the name of the thing you are GETDSK'ing.

EXTENSIONS:

- .DEC decimal mode
- .BIN binary mode
- .VGN vector general format
- .CST character set (automatically called by "text")
- .BLN present blend format
- .MAC macro format (for macros, not for pictures)

Note that .DEC, .VGN, .BLN come up on the screen. .BIN pictures are put into the non-displayed data structure (PUTLIB'd) and must be gotten by GETLIB to be seen. .BIN also preserves the setups of dials, etc for rotate, move, scale, etc., whereas the others do not.

EXAMPLES:
GETDSK WITCH
GETDSK WITCH. DEC
GETDSK WITCH. DEC([30, 10])
GETDSK DRAW. MAC
GETDSK ROMS. CST
GETDSK OHIO1. BLN
HELP FIX

**SYNTAX:** FIX PNAME

Fix freezes the position of a picture or the value of a PNAME's modifier according to the switch options. No switch fixes everything. Fix also removes dev assignments if any.

**SWITCHES:**

- **NONE** fixes all of the below
- **/R** fixes rotation
- **/S** fixes scale
- **/M** fixes move
- **/P** fixes pathmov
- **/I** fixes intensity
- **/C** fixes cutoff plane
- **/H** fixes hide feature
- **/O** fixes setorg

**EXAMPLES:**

- Fix/R coper
- Fix globe

HELP FLAP

**SYNTAX:** FLAP PNAME, EXPR1, EXPR2

Flap puts bounds on the rotation angle. The bounds are given by EXPR1 (the higher bound) and EXPR2 (the lower bound). Flap will settle faster if FLAP PNAME, O, O is typed first, followed by the FLAP command with the desired bounds.

**EXAMPLE:**

- Rotate wing, x, d0, d1, d2
- Flap wing, 100, -120

HELP FSOFF

**SYNTAX:** FSOFF EXPR1, EXPR2, EXPR3, ETC.

Fs off turns off the function switches corresponding to the expressions.

**EXAMPLE:**

- Fs off 1, 3, 5, 7, 9, 11, 13

This turns off the odd function switches. (Note that FS15 is not user programmable.)
HELP DASHES

SYNTAX: DASHES LNAME

DASHES CHANGES THE LNAME'S VECTORS TO DASH MODE.

SWITCHES:
NONE DASHES MODE
/R DOTS MODE (REALLY SHORT DASHES)

EXAMPLE: DASHES DIAMON

HELP DELETE

SYNTAX: DELETE LNAME

DELETES THE LNAME FROM CORE, REMOVES THE NAME AND RECLAIMS THE
STORAGE THE LNAME TOOK. DOES NOT PRESENTLY WORK WITH GROUPS.

SWITCHES:
NONE AS ABOVE
/D DELETES ON YOUR DISK AREA

EXAMPLES:
DELETE GLOBE
DELETE JSDRAW
DELETE/D WITCH, DEC

HELP DELPOI

SYNTAX: DELPOI

DELPOI DELETES THE LAST PREVIOUSLY PUT POINT IN AN OPENED LIST.
IT TAKES THE LNAME FROM THE OPENED COMMAND ISSUED.

EXAMPLE: <PUTPOI D0/16, D1/16, D2/16, 1
IF FS1, DELPOI
SKIP -2>
THIS CAUSES VECTORS TO BE DRAWN UNLESS FS1 IS PUSHED.

HELP DIRALL

SYNTAX: DIRALL

DIRALL GIVES THE USER THE DISK DIRECTORIES OF EVERYONE'S
AREAS.
HELP BYE

SYNTAX:   BYE

BYE LOGS USER OUT IF HE TYPES "YES" TO "CONFIRM". LOGIN MUST BE USED TO REENTER SYSTEM. BYE IS USEFUL FOR CHANGING DISK AREAS WHILE RUNNING THE GRAPHICS SYMBIOSIS SYSTEM.

HELP CLEAR

SYNTAX:   CLEAR

CLEAR SIMPLY CLEARS THE VT05 SCREEN.

HELP COPY

SYNTAX:   COPY LNAME1,LNAME2

COPY CAUSES LNAME2 (THE NEW NAME) TO SHARE THE DATA OF LNAME1 (THE OLD NAME). ANY COMMAND THAT DOES NOT MODIFY DATA LISTS (E.G. ROTATE, SCALE, MOVE, ETC) MAY BE THEN INDEPENDENTLY BE USED ON EITHER LNAME. COMMANDS THAT ALTER DATA LISTS (E.G. SMOOTH, SOFTROT) WILL MODIFY BOTH COPIES. COPY MAY BE USED TO REFLECT A PICTURE AROUND AN AXIS BY USING THE SINGLE DIMENSIONAL SCALE.

EXAMPLE:   COPY GLOBE,WORLD
            ROTATE GLOBE,X,0,0,D1,D2
            ROTATE WORLD,2,0,0
            SCALE WORLD,0,0

HELP COLOR

SYNTAX:   COLOR LNAME,EXPR

COLOR SETS UP THE LNAME FOR USE WITH THE COLOR WHEEL. THE EXPR MUST EVALUATE TO 0, 1, 2, 3, 4, OR THE LNAME WILL BE BLANDED. ZERO INDICATES ALL COLORS, THAT IS, WHITE. THE OTHER VALUES INDICATE ONE OF THE FOUR SEPARATE COLOR WHEEL COLORS, THE CHOICE OF WHICH MUST BE DONE AFTER THE COLOR WHEEL IS SPINNING. COLORS MAY BE MIXED BY USING THE COPY COMMAND AND SETTING THE COPY TO A DIFFERENT COLOR THAN THE ORIGINAL. IF THE IMAGE DOES NOT APPEAR TO HAVE PURE COLORS, TOO MUCH PICTURE IS BEING DISPLAYED AND THE NUMBER OF VECTORS BEING DRAWN MUST BE REDUCED IN THIS CASE.

EXAMPLES:   COLOR LEAF,1
            COLOR BUMP,4
HELP EXIT

SYNTAX: EXIT

EXIT IS A SUPER RETURN. IT ACTS LIKE A "C" WITHIN A MACRO AND RETURNS CONTROL TO "*" (I.E. YTOS) LEVEL.

HELP FILL

SYNTAX: FILL OUTLINE, FILL-LINES, FILL-CHARACTER, SPEED, DEGREES

FILL TAKES A LNAME AS AN OUTLINE, BUILDS A NEW LNAME COMPOSED OF THE FILL CHARACTER RANDOMLY PLACED WITHIN THE OUTLINE UNTIL FS 13 IS PUSHED. THE FILL CHARACTER SHOULD BE COMPOSED OF A SMALL NUMBER OF VECTORS IF MANY ARE NEEDED TO FILL THE OUTLINE. DEGREES (/P ONLY) INDICATES THE ANGLE AT WHICH THE FILL CHARACTER IS ROTATED. THE DEV FOR SPEED CONTROLS THE RATE AT WHICH THE FILL CHARACTER IS REPEATED.

SWITCHES:

- NONE FILL CHARACTER IS RANDOMLY ROTATED AS IT IS PLACED WITHIN OUTLINE.
- /P FILL CHARACTER IS PLACED AT ANGLE INDICATED BY DEGREES.

EXAMPLE:

FILL SQUARE, HAIR, CURL, D0
FILL/P HILL, FOREST, TREE, D0, 45

HELP FILMING

SYNTAX: FILMING

SETS SINGLE FRAMING MODE ("F" IS ENOUGH TO TRIGGER THIS COMMAND)
HELP GROUP

SYNTAX: GROUP PNAME1, PNAME2, GNAME

GROUPS PNAME1, PNAME2 AND EVERYTHING BETWEEN THEM INTO A GROUP NAMED GNAME. PNAMEs MUST BE DISPLAYED. PUTLIB/GETLIB SEQUENCES MAY BE USED TO ALTER GROUP STRUCTURE BY ELIMINATING OR ADDING PICTURES IF NECESSARY. ANY COMMAND WHICH WILL WORK ON PNAMEs WILL WORK ON GROUPED PICTURES.

EXAMPLE: GROUP AIRPLAN, PROP, PLANE

HELP HIDE

SYNTAX: HIDE

HIDE MAY BE USED TO HIDE THE REAR SURFACES OF SPECIALLY PREPARED 3-D SHAPED PICTURES. THE SURFACES MUST BE SET BY SETINT/S, DRAWN COUNTER-CLOCKWISE AS THEY ROTATE INTO VIEW, AND BE SHAPED BY SHADE /3, /4, /5 OR /6.

EXAMPLE: SHADE/5 STEEP, TOM, 4
SETINT/S TOM, 09
HIDE TOM

HELP INTERP

SYNTAX: INTERP OUTLINE, SHADE-LINES, EXPR

INTERP CREATES A NEW PICTURE BY RUNNING LINES BETWEEN THE VECTORS OF THE OUTLINE. THE NEW LINES ARE SPACED AS INDICATED BY THE EXPR. THE OUTLINE MUST HAVE AT LEAST ONE JUMP IN IT, BUT MAY HAVE MORE.

EXAMPLE: INTERP FLUFF, STUFF, 30

HELP LINES

SYNTAX: LINES LNAME

LINES PUTS LNAME'S VECTORS IN REGULAR DRAWING MODE. IT IS USED TO RECOVER FROM DASHES AND POINTS MODES.

EXAMPLE: LINES DIAMON
HELP LIST

SYNTAX: LIST

LIST CAUSES LINES OF MACROS TO BE ECHOED DURING EXECUTION. IT IS A USEFUL DEBUGGING TOOL. LIST IS SUPPRESSED BY XLIST.

HELP MOVE

SYNTAX: MOVE TNAME, ORIGIN

MOVES PNAME TO PLACE INDICATED BY ORIGIN. ORIGIN IS NEW X, Y, Z POSITION AND IS SPECIFIED BY DEV, DEV+1, AND DEV+2.

EXAMPLES:

MOVE WITCH, JS
MOVE TOM, F

THE FIRST EXAMPLE MOVES THE WITCH ACCORDING TO JX, JY, AND JZ.
THE SECOND EXAMPLE MOVES TOM ACCORDING TO VARIABLES F, G, AND H.

HELP OPENO

SYNTAX: OPENO LNAME

OPENO IS NECESSARY TO ALLOCATE THE CORE MEMORY NECESSARY TO CREATE A PICTURE. IT SETS THE LNAME FOR DELPOI AND PUTPOI ALSO.

EXAMPLES:

OPENO SAM
OPENO SA


HELP PATHMOV

SYNTAX: PATHMOV PNAME, LNAME, SPEED

PATHMOV MOVES PNAME TANGENTIALLY ALONG LNAME ACCORDING TO THE SPEED INDICATED BY A DEV.
THE FIGURE MUST BE ROTATING IF TANGENTIAL QUALITY IS DESIRED.
OTHERWISE, FIGURE WILL BE ORIENTED ALONG PATH JUST AS IT WAS ORIGINALLY.
THE FIGURE ALSO SHRINKS AS IT MOVES INTO THE Z AXIS.

EXAMPLES:

PATHMOV BFLY, TSINE, DO

HELP PENOFF

SYNTAX: PENOFF

PENOFF TURNS THE LIGHT PEN OFF.
HELP PENSEN

SYNTAX:    PENSEN LNAME

PENSEN MAKES A LNAME SENSITIVE TO LIGHT PEN HITS, AND TURNS OFF THE PREVIOUS PENSEN ASSIGNMENT. ONLY ONE LNAME MAY BE PENSEN'D AT A TIME.

EXAMPLE:   PENSEN TARGET

HELP POINTS

SYNTAX:    POINTS LNAME

POINTS CHANGES LNAME'S VECTORS TO DISPLAY ONLY THE ENDPOINTS.

EXAMPLE:   POINTS MOREY

HELP PRINT

SYNTAX:    PRINT DNAME. EXT [T30, X]

PRINT IS THE SAME AS TYPE BUT IT CREATES HARDCOPY ON THE DECWRITER INSTEAD OF DISPLAY ON THE VT05. PRINT RUNS AS A SUBJOB AND CANNOT BE EASILY STOPPED.

EXAMPLE:   PRINT DRAW. MAC
            PRINT WITCH. DEC [T30, 1]

HELP PUTDSK

SYNTAX:    PUTDSK DNAME. EXT

PUTDSK PUTS DNAME OUT TO YOUR AREA ON THE DISK IN THE FORMAT INDICATED BY THE EXTENSION.

EXTENSIONS:

.BIN    BINARY
.DEC    DECIMAL (DEFAULT)
.MAC    MACRO
.VGN    VECTOR GENERAL FORMAT

.BIN PRESERVES DEVICE ASSIGNMENTS AND OTHER MOTION INFORMATION. THE OTHERS DO NOT. .VGN AND .BIN MAY BE USED WITH ANY LNAME. .DEC MAY ONLY BE USED WITH 3-D NON-SHADDED DATA. YOU CANNOT WRITE ON ANYONE ELSE'S AREA.

EXAMPLES:  PUTDSK DRAW. MAC
            PUTDSK WITCH. VGN
            PUTDSK BLOB. BIN
            PUTDSK GLOBE
            PUTDSK WITCH. DEC
HELP PUTLIB

SYNTAX: PUTLIB PNAME

PUTS PNAME INTO NON-DISPLAYED DATA STRUCTURE. REMOVES PNAME FROM HIGHER GROUPS IF ANY. PNAME MUST BE DISPLAYED.
ALL COMMANDS WORK ON PUTLIB'D PICTURES EXCEPT THOSE EXPlicitLY STATED NOT TO BY THE COMMAND DESCRIPTION ITSELF.

EXAMPLE: PUTLIB OUTLINE

HELP PUTPOIN

SYNTAX: PUTPOIN X, Y, Z, K

PUTPOIN PUTS A POINT INTO THE LNAME INDICATED BY AN OPENO.
X, Y, Z ARE ANY EXPRESSIONS DETERMINING THE X, Y, AND Z COORDINATES OF THE POINT IN DECIMAL (RANGE -20-47 TO 2047). K INDICATES THE FOLLOWING:
K=0 DRAW THE VECTOR
K=1 MOVE TO A NEW PLACE BUT DO NOT DRAW (JUMP)
K=-1 END OF LIST
NOTE THAT (UNLIKE GETPOIN) IF K=-1, THE LAST X, Y, Z ARE IGNORED (I.E. THE LAST POINT SHOULD BE SPECIFIED)
PUTPOIN 0, 0, 0, -1
to terminate the list). also note that the first point automatically is stored as K=1

EXAMPLE: PUTPOIN D0/16, D1/16, D2/16, 1
PUTPOIN A, B, C, 0
PUTPOIN 0, 0, 0, -1

THIS WILL CREATE A PICTURE WITH A LINE DRAWN FROM THE INSTANTANEOUS POSITIONS OF D0, D1, AND D2 TO THE POSITIONS OF THE VARIABLES A, B, AND C. (THE /16 IS NECESSARY BECAUSE DIALS ARE READ LEFT JUSTIFIED.)

HELP READ

SYNTAX: READ DOSDEVICE

READ CHANGES THE READ DEVICE FOR TYPE AND PRINT TO THE NEW PDP-11 DIRECTORY DEVICE.

SWITCH:

None FOR TYPE JOB
/P FOR PRINT JOB

EXAMPLE: READ/P DF
PRINT GRASS, MAP(30, 4)
HELP REMARK

SYNTAX: REMARK * ANY TEXT YOU WANT FOR COMMENTS
UNTIL A DELIMITER IS SEEN!* 

THE FIRST CHARACTER AFTER THE COMMAND IS TAKEN AS A
DELIMITER AND ALL THE FOLLOWING TEXT UNTIL THE DELIMITER IS SEEN AGAIN
IS TAKEN AS COMMENTS. REMARK IS USEFUL FOR DOCUMENTING MACROS.

EXAMPLE: REMARK % THIS MACRO MAKES FLIES FLY ON DIAL 0, AND THE JOYSTICK

HELP RENAME

SYNTAX: RENAME ANAME1, ANAME2

RENAME REPLACES ANAME1 WITH ANAME2 IN THE NAME TABLE.
IT IS USEFUL IF A NAME HAS TO BE CHANGED TO BRING ANOTHER COPY OF
A CORE RESIDENT PICTURE INTO CORE, OR TO PUTDSK A MODIFIED
PICTURE WHOSE NAME IS ALREADY ON THE DISK.

SWITCHES:
NONE AS ABOVE
/D RENAMES DISK FILE ON YOUR AREA

EXAMPLES:
GETDSK HITCH
RENAME HITCH, OLDLADY
GETDSK HITCH
HOW THERE ARE TWO COPIES OF HITCH IN MEMORY.

RENAME/D HITCH, DEC, LADY, DEC
HITCH, DEC IS NOW CALLED LADY, DEC ON THE DISK.

HELP RESTART

SYNTAX: RESTART

RESTART RE-INITIALIZES THE ENTIRE GRASS SYSTEM.

HELP RESUME

SYNTAX: RESUME

USED ONLY TO RETURN FROM VT05 INPUT MODE TO A MACRO IN EXECUTION
OR TO A FS15-INTERRUPTED GETDSK.

EXAMPLE: *GETDSK GLOBE
(PUSH FS15)
*ROTATE GLOBE,X,0
RESUME
(FINISH GETDSK)
* ETC
ALSO SEE "WAIT" WHICH IS USED FOR GETTING VT05
INPUT WHEN IN A MACRO.
HELP ROTATE

SYNTAX:  

ROTATE FRAME, AXIS, SPEED, TILT, ORIGIN


SWITCHES:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/N</td>
<td>None</td>
</tr>
<tr>
<td>/S</td>
<td>Fast Rotation</td>
</tr>
<tr>
<td>/D</td>
<td>Dev Indicates Angle Of Rotation Instead Of Speed</td>
</tr>
</tbody>
</table>

EXAMPLES:

- ROTATE PROP, X, 09
- ROT/S GLOVE, A
- ROT/D CUBE, Z, 08
- ROTATE GLOVE, X, 09, 01
- ROTATE BLADE, Z, 00, 01, 02
- ROTATE GLOVE, X, 09, 01, 02
- ROTATE/D FRED, K, L, F

HELP SCALE

SYNTAX:  

SCALE FRAME, FACTOR

Scales Frame By Factor Indicated By A Dev.

SWITCHES:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/N</td>
<td>None</td>
</tr>
<tr>
<td>/X</td>
<td>Uses Hardware Scale Which Scales To Picture's 0, 0, 0 As Modified By /HOF, /ROTATE, Etc.</td>
</tr>
<tr>
<td>/Y</td>
<td>Scales Along X Axis (Single Dimensional Scaling).</td>
</tr>
<tr>
<td>/Z</td>
<td>Scales Along Y Axis</td>
</tr>
<tr>
<td>/R</td>
<td>Scales Along Z Axis</td>
</tr>
<tr>
<td>/N</td>
<td>Single Dimensional Scaling On Dev, Dev+1 And Dev+2 Scaling With Switches Also Allows The Origin Of Scale To Be Reset By Using SETORG Command</td>
</tr>
</tbody>
</table>

EXAMPLES:

- SCALE TON, 00
- SCALE/X NOS, 09
- SCALE/Z SPIRAL, Z

The Last Line Takes J For X Scale, K For Y Scale And L For Z Scale.
HELP SETCUT

SYNTAX: SETCUT, PNAME, VALUE

SETCUT SETS THE CUTOFF PLANE ACCORDING TO THE VALUE SPECIFIED BY A DEV. IF DEV IS SET TO BE SLIGHTLY NEGATIVE, THEN VARYING THE INTENSITY SET BY SETINT WILL MOVE THE CUTOFF PLANE WITH RESPECT TO THE Z-AXIS.

EXAMPLE: SETCUT GLOBE, .DO

HELP SETDELA

SYNTAX: SETDELA DELAY

THE DELAY, INDICATED BY A DEV, IS USED TO SLOW DOWN GETDSK FOR .DEC AND .VGN MODES. SETDELA REMAINS SET FOR THE REST OF YOUR SESSION OR UNTIL RESET BY, FOR EXAMPLE A=0

EXAMPLE: SETDELA .DO

HELP SETINT

SYNTAX: SETINT PNAME, VALUE

SETINT SETS THE INTENSITY OF THE PNAME AS INDICATED BY THE DEV USED FOR VALUE.

SWITCHES:

NONE CHANGES INTENSITY FOR ENTIRE PNAME
/S FOR USE WITH 3-D SHADED OBJECTS ONLY. FS13 IS USED TO GO FROM SURFACE TO SURFACE, SETTING THE INTENSITY ON THE DEV INDICATED.

EXAMPLE: SETINT GLOBE, .DO SETINT/S CUBE, .D9
HELP SHADE

SYNTAX: SHADE OUTLINE, SHADE-LINES, SPACING, DEGREES

SHADE HAS MANY OPTIONS. THE OUTLINE IS ALWAYS A 3-D DATA
LNAME (WHICH SHOULD BE PLANAR FOR THE EXPECTED EFFECT). SHADE LINES
ARE BUILT IN A NEW LNAME ACCORDING TO THE SWITCH INDICATED. SPACING
IS AN EXPRESSION WHICH CONTROLS HOW FAR THE LINES ARE SPACED APART
(3 IS THE DEFAULT). DEGREES INDICATES THE ANGLE OF THE SHADING
LINES (/P, /N, /3, AND /4 MODES ONLY)

SWITCHES:

NONE

/P PARALLEL 2-D SHADING. BETTER QUALITY SHADING THAN AUTO-
INCREMENT MODE. (TWO WORDS PER VECTOR.)

/N 2-D AUTO-INCREMENT SHADING WHICH OBSERVES JUMPS. (E.G.
IT WILL SHADE A DONUT CORRECTLY WHEREAS NO-SWITCH MODE
WILL NOT). (ONE WORD PER VECTOR PLUS JUMPS.)

/X PARALLEL /N MODE (TWO WORDS PER VECTOR PLUS JUMPS.)

/X3 3-D SHADING WHICH LEAVES OUTLINE VISIBLE. INCLUDES
HIDDEN SURFACE AND INTENSITY INFORMATION. (THREE
WORDS PER VECTOR.) ALL THREE-DIMENSIONAL SHADING
REQUIRES CLOSED SURFACES IN A PLANE WITH THE ENDPOINTS
REPEATED IN THE CASE OF ADJACENT PLANES. IF HIDDEN
SURFACE EFFECT IS DESIRED IN THE CONVEX CASE, SURFACES
MUST BE SPECIFIED IN A COUNTER-CLOCKWISE ORDER AS THE
OBJECT IS ROTATED INTO VIEW.

/X4 LIKE /X3 EXCEPT THAT THE OUTLINE IS REMOVED.

/X5 OPTIMIZED /X3. THIS IS THE ONE NORMALLY USED.

/X6 OPTIMIZED /X4.

EXAMPLES:

SHADE /X S W I T C H , O L D L A D Y

SHADE /P BOX, F L A N E , 1 0 , 1 5

SHADE BLOB, N E N E L B O B , 1 7

HELP SKIP

SYNTAX: SKIP EXPRESSION

SKIP IS USED TO TRANSFER CONTROL WITHIN A MACRO FOR LOOPS.
EXPRESSION MUST EVALUATE TO A NUMBER WHICH, IF NEGATIVE, SKIPS BACKWARDS
THAT NUMBER OF LINES; IF POSITIVE, CONTROL SKIPS FORWARD THAT
NUMBER OF LINES. NUMBERS LARGER THAN POSSIBLE WITHIN THE
MACRO WILL RESULT IN TRANSFER TO THE FIRST STATEMENT OF THE PROGRAM IF
NEGATIVE, OR PAST THE LAST (AND THEREFORE EFFECTIVELY A RETURN) IF POSITIVE.

EXAMPLE:

<X=X+I
PROMPT X
IF FS1=O, SKIP -2>

THIS EXAMPLE WILL INCREMENT X AND PRINT IT UNTIL FS1 IS PUSHED.

HELP SMOOTH

SYNTAX: SMOOTH LNAME, NUMBER

SMOOTH USES A MODIFIED QUADRATIC SMOOTHING TECHNIQUE TO
SMOOTH THE 3-D LNAME THE NUMBER OF TIMES INDICATED BY AN EXPRESSION.

EXAMPLES:

SMOOTH PATH, 7

SMOOTH GLOBE, 1 1 / 8 - A * D 0

*
HELP SOFT

SYNTAX: SOFTROT LNAME

SOFTROT TAKES THE VARIOUS HARDWARE MODIFICATIONS TO THE LNAME (SPECIFICALLY ROTATION, MOVING, AND SCALING) AND SOFTWARE ALTERS THE DISPLAY LIST TO GIVE ACTUAL VECTORS REPRESENTING THESE CHANGES. SOFTROT THEN DOES A RESET TO HALT FURTHER HARDWARE ROTATION, MOVING, SCALING, UNTIL THE USER RE-ROTATES, ETC. (IF HE DOES). A PUTDISK WOULD THEN PUT THE TRANSFORMED LIST ONTO THE DISK. TOO MANY SOFTROT'S WILL SLOWLY DESTROY THE PICTURE DUE TO ROUNDOFF ERRORS.

EXAMPLE: ROT GLOBE,Y,10
SCALE GLOBE,D1
SOFTROT GLOBE

HELP TEXT

SYNTAX: TEXT CNAME,LNAME

+ONE LINE OF ANY TEXT EXCLUDING "<" AND ">

TEXT AUTOMATICALLY CALLS THE CHARACTER SET CNAME INTO CORE AND IF NECESSARY AND CREATES AN LNAME CONSISTING OF CHARACTER SUBROUTINE CALLS. DO NOT DELETE THE CNAME, OTHERWISE THE LNAME WILL INSTANTLY CAUSE THE SYSTEM TO CRASH. EACH LNAME REPRESENTS ONE AND ONLY ONE LINE OF TEXT. THE AVAILABLE CHARACTER SETS ARE AS FOLLOWS:

R0H2 DOUBLE LINE ROMAN FONT (UPPER AND LOWER, COMPLETE PUNCTUATION)
SCR SINGLE LINE SCRIPT FONT (UPPER AND LOWER)
SCR2 DOUBLE LINE SCRIPT FONT (UPPER AND LOWER)
MAP SELECTED MAP SYMBOLS
ITALL LOWER CASE ITALIAN GOTHIC FONT
ITALU UPPER CASE ITALIAN GOTHIC FONT
ENGL LOWER CASE ENGLISH GOTHIC FONT
ENGB UPPER CASE ENGLISH GOTHIC FONT
RUSL LOWER CASE CYRILLIC FONT
RUSB UPPER CASE CYRILLIC FONT

TO INDICATE LOWER CASE IN CHARACTER SETS WITH BOTH UPPER AND LOWER CASE, YOU MUST TYPE A "\" BEFORE EACH LOWER CASE CHARACTER TO BE USED. A DOUBLE "\" (I.E. "\") IS USED AS A SHIFT LOCK. IF "\" IS USED, "\" MUST BE USED TO INDICATE AN UPPER CASE LETTER. "\" WILL SHIFT LOCK TO UPPER CASE AGAIN.

EXAMPLES: TEXT ITALL,TOM
THIS IS AN EXAMPLE

TEXT SCR2,SAM
GRAPHICS\SYMBIOSIS\SYSTEM

NOTE THAT SPACES MUST BE ENTERED AS UPPER CASE CHARACTERS. PUNCTUATION IN THOSE SETS WHERE AVAILABLE, IS INDICATED BY THE SAME PUNCTUATION CHARACTER BEING TYPED. MAP AND RUSL\RUSB HAVE SPECIAL CORRESPONDANCES AVAILABLE FROM THE SYSTEMS PROGRAMMERS.
HELP TREE

SYNTAX: TREE

TREE GIVES THE USER A DIAGRAM OF HIS DATA STRUCTURE. IT LISTS THE PICTURES AND GROUPS IN THE DISPLAYED STRUCTURE BY NAME, INDICATING GROUPING LEVELS AND HIERARCHIES BY TABS.

TREE ALSO CHECKS OUT THE DATA STRUCTURE AND INFORMS THE USER BY AN ERROR MESSAGE IF HIS DATA STRUCTURE CONTAINS IMPROPER GROUPS (GROUPS ACROSS LEVELS, FOR INSTANCE) WHICH WOULD CAUSE PROBLEMS IF COMPOUND TRANSFORMATIONS WERE APPLIED. IF THE ERROR MESSAGE IS SEEN, THE USER SHOULD PUTLIB ALL HIS PICTURES AND THEN GETLIB THEM ONE BY ONE, GROUPING WHERE NEEDED, AND CHECKING THE INTEGRITY OF THE STRUCTURE BY USING TREE PERIODICALLY.

HELP TYPE

SYNTAX: TYPE DNAME. EXT{30, X}

TYPE TYPES THE DNAME FROM THE AREA INDICATED BY "{30, X}"

THE DNAME IS TYPED ON THE VT05 AND MAY BE ABORTED BY TYPING "C". IF NO "{30, X}" IS INDICATED, THE DEFAULT IS TO YOUR AREA.

EXAMPLES: TYPE DRAW. MAC
           TYPE HITCH. DEC

HELP UNFILM

SYNTAX: UNFILM

RESETS TO NORMAL RUN MODE AFTER FILMING WAS USED. (*U* IS ENOUGH TO TRIGGER THIS COMMAND.)

HELP WAIT

SYNTAX: WAIT

WAIT CAUSES THE MACRO TO WAIT FOR THE USER TO TYPE ON THE VT05. IT INHIBITS THE EXECUTION OF THE MACRO UNTIL THE USER TYPES "RESUME."

WAIT IS USEFUL FOR INTERACTION WITH THE USER DURING THE USE OF A LONG MACRO.

EXAMPLE:

<....
PROMPT "TYPE HELP FOLLOWED BY THE COMMAND NAME"*
PROMPT "IF THERE IS A COMMAND THAT YOU DO NOT BROK."*
PROMPT "TYPE RESUME WHEN YOU WISH TO CONTINUE WITH THIS"
PROMPT "SELF-HELP MACRO"
WAIT
....>
HELP WARP

SYNTAX: WARP LNAME,DEV1,DEV2

WARP CHANGES THE SHAPE OF LINE SEGMENTS ACCORDING TO THE POSITION OF THE DEV'S. WARP IS MUCH EASIER TO USE IF THE WARP MACRO IS USED. THIS MACRO GETS A CURSOR, SETS THE DIALS FOR MOVING THE CURSOR AND ALLOWS THE USER TO SWITCH WARP MODES. THE LNAME MUST NOT BE IN MOTION OR BE SCALCED (IF IT IS, USE SOFTROT). THE WARP MACRO OVERLAY IS FAIRLY SELF-EXPLANATORY.

SWITCHES:
- NONE
- /B WARP'S LINE BUT PRESERVES ENDPOINTS
- /H WARP'S LINE BETWEEN CHOSEN POINTS (USE WARP MACRO)
- /M TRANSLATES PART OF PICTURE BETWEEN JUMPS TO CURSOR POSITION

EXAMPLE: USE WARP MACRO

HELP WRITE

SYNTAX: WRITE DOSDEVICE

WRITE IS THE SAME AS READ EXCEPT THAT IT CHANGES THE PUTDSK DEVICE.

HELP COMPIL

SYNTAX: COMPILE MNAME1,MNAME2

COMPILE RESOLVES :LABEL PSEUDO-LABELS IN MNAME1. IF A MACRO HAS PSEUDO-LABELS, COMPILE MUST BE USED BEFORE IT CAN BE EXECUTED. PSEUDO-LABELS ARE USED TO ELIMINATE THE TASK OF COUNTING LINES FOR SKIP ARGUMENTS IN LONG MACROS. MNAME1 IS AUTO-DELETED. IF MNAME1 IS CALLED FROM THE DISK, OR PUTDSK'D BEFORE IT IS COMPILE'D IT MUST HAVE THE EXTENSION .MCS.

EXAMPLE: GLOB:<IF FS1,SKIP XFLUFF SKIP -1 XFLUFF: DIRCOR> COMPIL GLOB,CGLOB

CGLOB NOW LOOKS LIKE:
CGLOB:<IF FS1,SKIP 2 SKIP -1 DIRCOR>

GLOB IS AUTO-DELETED AND CGLOB CAN BE EXECUTED.
Appendix B. System Configuration

Computer:

PDP-11/45
48K Memory, 16K MOS, 32K Core
RF and RK Disk and Cartridge Drives
Dual DECTape Drives
Paper Tape Punch and Reader
DECreater and VT05

Graphics Equipment:

Vector General 3-D Rotation Display
With Joystick, Ten Dials, Function
Switches, Light Pen, and Analog
Hardware for Rotation, Scaling, Intensity,
Translation, Depth Cueing and Z-Axis Cutoff.
Scientific Accessories Corporation 3-D Graf-Pen.
Arriflex 16mm Motion Picture Camera with
Animation Motors.
Appendix C. Program Examples

This Appendix gives several examples of usage of the Graphics Symbiosis System. Ten macros which represent a range of capabilities are reproduced here. In several cases, the user interaction with the macro at run-time is given in addition to the text of the macro. Most of the macros have pseudo-labels which are resolved by a one-pass processor command "COMPILE." The examples start with a simple macro and then progress to more complex examples. Appendix A can help in deciphering command syntax if there is difficulty following the code. Remember that names may be abbreviated to just enough letters to insure uniqueness so that "SHADE" is often coded as "SH," for example. The macros will be referenced below as "Macro C-1, Macro C-2,..."

"ROW3" (Macro C-1) is a simple user-written macro that sets up scaling and translation for any two pictures. The user thus has created a command for himself which does six repetitious lines of code in one call and two name responses. This is the type of macro that often makes the system easier to use and can be quickly written and tailored to any user's needs. Thus it is a very simple illustration of the use of the system's extensibility to add to the system's habitability.
"WARP" (Macro C-2) is a macro which aids usage of the "WARP" command. It calls in a cursor whose motion is set to correspond to the location of the segment to be warped, and lets the user switch between the modes of warping by pushing function switches. It is an example of how the use of a complex primitive command is made easier and more flexible by embedding it in a macro. A picture of the function switches and overlay used with this macro is also included.

"THICKR" (Macro C-3) is a data generation macro. It uses "GETPOIN/PUTPOIN" sequences to modify x,y,z coordinates as the user specifies. This macro illustrates a procedure which would simply be too time consuming and tedious to do without macros. An example of the output of "THICKR" can be seen in Figure 7. In the example of user interaction following the macro, system typed output is underlined. The user need only type the words and numbers that are not underlined. The core directory commands ("DIRC") show the size of the "BUNNY" in words before and after the use of "THICKR."

"DBLPOI" (Macro C-4) is a simple data manipulation macro written by an art student. It is included here to document the ease of writing useful macros. "DBLPOI" places a point in the middle of every line segment in the drawing and thus doubles the original number of points. This macro is used to saturate a path to make the path following motion
smoother if necessary, and also to create more points if "WARP" must be used on a long line segment. Variations like tripling the number of points or selectively placing the points in the middle of selected line segments can be done within a similar framework with more code.

"MERGE" (Macro C-5) allows users to concatenate data lists and thereby build up pictures from simpler elements. If the simpler elements never need to be individually manipulated, this procedure is more efficient than grouping the elements. An example of user interaction with "MERGE" is given after the macro. Note the use of a computed "SKIP" which allows the correct ordinal suffix to be attached to the printout.

"LPDRAW" (Macro C-6) is a sophisticated macro which allows the user to draw with the lightpen. It operates sufficiently fast to keep ahead of the user—in other words, the user is not aware of the time it takes to process and record a point. "JSDRAW" and "SPDRAW" are similar macros not shown here which permit drawing with the joystick and sonic pen respectively. To delete points, "LPDRAW" calls an external macro "JSDDEL" which is common to all the drawing macros. Note that all the space taken up by the code and tracking cross is reclaimed by the "DELETE's" at the end. Several new commands were added ("GETHIT", "PENSEN", etc.) as well as a new "IF" condition: "IF LP." These additions
can be used in other macros that could make use of lightpen input for data identification and manipulation, for example.

"SWDEMO" (Macro C-7) is one of several demonstration macros. It sets up the wing flapping for the shaded swallow image that is found in several of the photographs. The last section of code aids the user in coordinating the flapping of the two wings together. Use of the panic button is also shown. Note how confusing the "SKIP's" can become even in a simple case when pseudo-labels are not used.

"TRASCL" (Macro C-8) allows the user to software scale his data by integral factors. Combining this macro with hardware scale give a facility whereby any picture may be transformed to any size. In addition, a software translate may be indicated with this macro. The example of user interaction following this macro shows doubling of the x and y coordinates and tripling of the z coordinates, and a y-translation of 500 raster units (approximately two inches).

"REFLEC" (Macro C-9) is a macro which will reflect the image around the specified axis or axes and concatenate this reflection with the original. It is particularly useful in the case of symmetric figures for which the artist need only digitize or draw one-half of the data, using "REFLEC" to get the whole image. If the data is symmetric to more than one axis, this macro may be used to build a figure from one
fourth or even one eighth of the intended image.

Last is the "EDITOR" (Macro C-10) which illustrates some of the alphanumeric handling possible in the Graphics Symbiosis System. The "$A" matching is a variant of the keyboard substring matching which allows matches of any alphanumeric strings in macros, decimal modules or any other alphanumeric data, as well as keyboard input. A search command could easily be added. This macro shows that an editor of fair complexity can be written for the system within the system by adding a few commands which will prove useful for general text manipulation as well.
Macro C-1

TYPE ROW3. MAC

CLEAR

FROMP "ROW3 - MOV X,Y SC X,Y SC BOTH"

FROMP "FIRST FEATURE: ";

INPUT $A

FROMP "SECOND FEATURE: ";

INPUT $B

MOV $A, D0

SC/A $0, D2

SC $A, D4

MOV $B, D5

SCALE/A $B, D7

SCALE $B, D9

DELETE ROW3


Macro C-2

TYPE HARP. NCS

CLEAR

FROMP "HARP - 3 OPTIONS - USE GRASS OVERLAY &4"

FROMP "CURSOR IS ON D8,D9"

FROMP "WHAT IS TO BE HARPED ";

INPUT $A

FSOFF 0,1,2,9

FS01 16

DELETE CURSOR

MOV CURSOR, 16

XLOOP IF F50 EO 1, SKIP XHARP

IF F51 EO 1, SKIP XHARP

IF F52 EO 1, SKIP XHARP

IF F59 EO 1, SKIP XEXIT

SKIP XLOOP

XHARP HARP/H #A,D8,D9

SKIP XLOOP

XHARP HARP #A,D8,D9

SKIP XLOOP

XHARP HARP/B #A,D8,D9

SKIP XLOOP

XEXIT DELETE CURSOR

DELETE HARP

*
The function switch assignments for the "WARP" macro. It is capable of producing a wide range of linear distortions and translations of figures.
2D TO 3D IMAGE PROJECTOR

FROMP "THICKR--2D THICKNESS UTILITY"
FROMP "ENTER THE FOLLOWING"
FROMP "PICTURE TO BE THICKENED"
INPUT $A
GETPOI 4,A,1,V,V,V
A=0
FROMP "NUMBER OF PROJECTIONS"
INPUT I
J=1
FROMP "IMAGE SPACING"
INPUT S
FROMP "X SKIN MAGNITUDE"
INPUT O
FROMP "Y SKIN MAGNITUDE"
INPUT P
OPEN THKTMP
T=0
W=0
D=0
N=1
ZLOOP1 GETPOI 4,A,N,X,Y,Z,K
IF K NE -1, SKIP ZNOSET
G=-1
K=0
ZNOSET PUTPOI X+T,Y+D,Z+W,K
N=N+1
IF G NE -1, SKIP ZLOOP1
T=T+O
W=W+S
D=D+P
I=I-1
G=0
V=N
N=1
IF I GT 6, SKIP ZLOOP1
FROMP $J, "VECTORS TOTAL"
FROMP "FINAL X="X+T-0," FINAL Y="Y+D-P," FINAL Z="Z+W-S
PUTPOI 0,0,0,1
DELETE $A
RENAME THKTMP,$A
FROMP "NO ERRORS DETECTED"
DELETE THICKR
LOG 30.4
PASSWORD: NICE

WELCOME TO PDP-11 GRASS
GETC BUNNY

CORE DIRECTORY

<table>
<thead>
<tr>
<th>PICTURE SIZE</th>
<th>FLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUNNY</td>
<td>268</td>
</tr>
</tbody>
</table>

768. WORDS USED IN 1. PICTURES.
48992. WORDS FREE IN 4. FRAGMENTS.

CALL THICKR
THICKR - ED THICKNESS UTILITY.
ENTER THE FOLLOWING:
PICTURE TO BE THICKENED - BUNNY
NUMBER OF PROJECTIONS - 35
IMAGE SPACING - 2120
X SCALE MAGNITUDE - 2100
Y SCALE MAGNITUDE - 20
4450. VECTORS TOTAL.
FINAL X = 230, FINAL Y = 730, FINAL Z = 450.
NO ERRORS DETECTED

CORE DIRECTORY

<table>
<thead>
<tr>
<th>PICTURE SIZE</th>
<th>FLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUNNY</td>
<td>3616</td>
</tr>
</tbody>
</table>

3616. WORDS USED IN 1. PICTURES.
2744. WORDS FREE IN 4. FRAGMENTS.
CLEAR
REM "DOUBLE POINT MACRO--TIM LIPETZ"
REM "DBLPoi DOUBLES NUMBER OF POINTS IN FIGURE"
REM "ENTER NAME OF FIGURE"
INPUT $A
REM "ENTER NEW NAME"
INPUT $B
OPENO $E
N=0
ZINCRR N=N+1
REH /GET THE FIRST POINT OF PAIR/
GETPOI $A,N,A,B,C,K
REH /OUTPUT THIS POINT/
PUTPOI A,B,C,K
REH /GET THE SECOND POINT OF PAIR/
GETPOI $A,N+1,D,E,F,L
REH /CHECK IF END OF DATA (-1) OR BLANKED LINE (1)/
IF L= -1, SKIP ZEXIT
IF L= 0, SKIP ZG
REH /IF BLANKED LINE, GET NEXT POINT/
GETPOI $A,N+2,S,T,U,V
IF V= -1, SKIP ZG
SKIP ZINCRR
ZG G=I+D
N=B+D
I=C+F
PUTPOI G/2,H/2,I/2,0
SKIP ZINCRR
ZEXIT PUTPOI 0,0,0,-1
*
Macro C-5

NEW

(Identification GRSUTL-101-V01)
G.B. Hoersdorff

*. THIS PROGRAM MERGES INCORE PICTURES *. 
REM /GET THE NAME OF THE PICTURE TO MERGE ALL THE OTHERS INTO /
PRINT "ENTER THE NAME OF THE OUTPUT PICTURE ");
INPUT \$A
REM /OPEN THAT NAME DYNAMICALLY /
OPEN \$A
REM / 'F' IS THE NUMBER OF FILES MERGED SO FAR /
F=1
REM / THIS NEXT ROUTINE DECIDES WHAT CONTRACTION TO USE ON /
THE PRINTOUT FOR THE NUMBER OF EACH FILE THE USER IS NOW MERGING. /
THIS DEMONSTRATES THE USE OF THE "SKIP" INSTRUCTION /
AS A FORTRAN-LIKE COMPUTED GOTO STATEMENT. /
XLOOP1 PRINT "ENTER THE ";F
S=F
REM / IF HE IS MERGING THE FOURTH FILE OR GREATER THE CONTRACTION IS 'TH' /
IF S GT 4, S=4
S=S+2
S=S-1
SKIP S
PRINT "ST";
PRINT XCONT
PRINT "ND";
PRINT XCONT
PRINT "RD";
PRINT XCONT
PRINT "TH";
XCONT PRINT " FILE TO BE MERGED ");
REM / ASK FOR THE NEXT PICTURE TO MERGE /
INPUT \$B
N=1
T=0
REM / THIS LOOP COPIES THE CURRENT PICTURE TO BE MERGED /
INTO THE OUTPUT PICTURE /
XLOOP3 GETPOI \$B, N, X, Y, Z, K
IF K NE -1, SKIP XNOSET
T=1
K=0
XNOSET PUTPOI X, Y, Z, K
N=N+1
IF T NE 1, SKIP XLOOP3
F=F+1
REM / COME HERE WHEN CURRENT PICTURE IS COPIED INTO OUTPUT PICTURE /
AND SEE IF THE USER HAS MORE TO MERGE /
PRINT "ANOTHER FILE (1=YES, ELSE NO) ");
INPUT N
IF N EQ 1, SKIP XLOOP1
REM / COME HERE WHEN THE USER SAYS THAT HE IS DONE /
PUTPOI 0,0,0,-1
PRINT \$A, " IS THE NEW FILE";
PRINT "NO ERRORS DETECTED"
REM / DELETE THE MACRO ITSELF /
DELETE MERGE
EXIT
LOG
O.S.U. COMPUTER GRAPHICS *PRODUCTION VERSION* 64-FEB-73 GRASS VIA 204

PASSWORD: NICE

WELCOME TO PDP-11 GRASS

GETC 'PROP
GETC ROBIN
GETD SBOD
GETD SWING1
GETD SWING2
CALL HERGE

ENTER THE NAME OF THE OUTPUT PICTURE 2BIGER

ENTER THE 1. ST FILE TO BE MERGED 'PROP
ANOTHER FILE (1=YES, ELSE NO) ?1
ENTER THE 2. ND FILE TO BE MERGED 'ROBIN
ANOTHER FILE (1=YES, ELSE NO) ?1
ENTER THE 3. RD FILE TO BE MERGED 'SBOD
ANOTHER FILE (1=YES, ELSE NO) ?1
ENTER THE 4. TH FILE TO BE MERGED 'SWING1
ANOTHER FILE (1=YES, ELSE NO) ?1
ENTER THE 5. TH FILE TO BE MERGED 'SWING2
ANOTHER FILE (1=YES, ELSE NO) ?5
BIGER IS THE NEW FILE

NO ERRORS DETECTED

C
CLEAR
RENAME
*  (IDENTIFICATION GRSUTL-99-Y01)
G. B. HOERSDORF 25-JAN-72
*
GETDSK JSDDDEL MAC[31,31]
GETCON LPCURS
POINTS LPCURS
PENSEN LPCURS
FSOFF 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
PROMP "LPDRAW--LIGHT PEN DRAW UTILITY--USE OVERLAY #3"
X=0
Y=0
Z=0
MOVE LPCURS,X
PROMP "WHAT IS THE DRAWING'S NAME?";
INPUT $A
OPEN $A
S=16
R=17
K=0
N=0
0=0
P=0
XLOOP1 IF FS1 EQ 1, K=1
IF FS2 EQ 1, SKIP XIBAND
IF FS3 EQ 1, SKIP XDEL
IF FS4 EQ 1, SKIP XLXX
IF FS5 EQ 1, SKIP XKY
IF FS6 EQ 1, SKIP XL2
IF FS7 EQ 1, SKIP XEXIT
IF LP, SKIP XIHIT
SKIP XLOOP1
WHIT GETHIT D,A,B,C,D
A=A+S
B=B+S
C=C+S
IF N EQ 0, X=X+A
IF 0 EQ 0, Y=Y+B
IF P EQ 0, Z=Z+C
IF FS0 NE 1, SKIP XLOOP1
PUTPOI X/S, Y/S, 2/S, R
FSOFF 1
K=0
J=0
J=J+32760
J=J/2
XLOOP2 J=J-200
IF J GT 0, SKIP XLOOP2
FSOFF 0
SKIP XLOOP1
XDEL DELPOI
DO JSDDDEL
FSOFF 3
SKIP XLOOP1
XLXX IF N EQ 0, SKIP XSETX
N=0
FSOFF 12
XOUTX DO JSDDEL
FSOFF 4
SKIP XLOOP1
XSETX N=1
FSON 12
SKIP XOUTX
XLY, IF P EQ 0, SKIP XSETY
O=0
FSOFF 13
XOUTY DO JSDDEL
FSOFF 5
SKIP XLOOP1
XSETY O=1
FSON 13
SKIP XOUTY
XLY2 IF P EQ 0, SKIP XSET2
P=0
FSOFF 14
XOUT2 DO JSDDEL
FSOFF 6
SKIP XLOOP1
XSET2 P=1
FSON 14
SKIP XOUT2
XAND IF LP, SKIP XEXIT
SKIP XAND
XEXIT GETHIT D,A,B,C,D
A=A+S
B=B+S
C=C+S
IF N EQ 0, X=X+A
IF O EQ 0, Y=Y+B
IF P EQ 0, Z=Z+C
PUTPOI X/S, Y/S, Z/S, K
FSOFF 2
L=R
XLOOP3 L=L-1
IF L NE 0, SKIP XLOOP3
IF FS2 NE 1, SKIP XLOOP1
DELETE LPPOI
SKIPP XAND
XEXIT PUTPOI 0, 0, 0; -1
FSOFF 4A, * CREATED*
FSOFF 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
DELETE LPCURS
DELETE JSDDEL
DELETE LPDRAW
EXIT
TYPE SWDEMO.MAC

PROM "SHALLOW FLYING MACRO"
GETD SB0D, DEC[20, 4]
SH/3 SB, SSBOO
GETD SSH1, BIN[20, 4]
GETD SSH2, BIN[20, 4]
GETL SSH1
GETL SSH2
DEL SB0D
GROUP SSBOO, SSH2, BIRD
ROT/D BIRD, V, D1
A=-32767
B=32752
C=1296
D=0
E=15000
ROT SSH1, V, D0, A, E, C
ROT SSH2, V, D0, A, E
FLAP SSH1, 0, 0
FLAP SSH2, 0, 0
PROM "FLAP SPEED ON D0"
PROM "BIRD ROT/D ON D1"
PROM "PUSH FS1 TO START OTHER WING FLAPPING"
A=A
IF FS1=0, SKIP -1
FLAP SSH2, -E, E
PROM "IF FLAPPING INCORRECT, PUSH FS1 TO STOP WING"
PROM "WAIT FOR WING TO SETTLE"
PROM "THEN PUSH FS2 TO START WING FLAPPING AGAIN"
PROM "REPEAT IF NECESSARY"
PROM "EXIT WITH PANIC BUTTON PUSH WHEN OK"
FSOFF 1, 2
IF FS1, SKIP 3
IF FS2, SKIP 5
SKIP -2
FLAP SSH2, 0, 0
FSOFF 1
SKIP -4
FLAP SSH2, -E, E
SKIP -8
TYPE TRASCL.MCS

CLEAR
REH * (IDENTIFICATION GRSUTL-061-801)
G.B. HOERSDORF

* THIS PROGRAM SCALES DATA UP VIA SOFTWARE
*
PROMP "TRANSCL--SOFTWARE SCALE/TRANSLATE UTILITY"
PROMP "ENTER THE PICTURE TO BE WORKED ON"
INPUT $A
D=0
REH / THIS NEXT LINE IS TO SEE IF WHAT HE TYPED IS IN CORE AND
IF IT IS NOT IT WILL DIE AND HE NEED NOT ENTER ANY MORE DATA/
GETPOI #A, 1, F, F, F, F
PROMP "ENTER 'X' SCALE FACTOR "
INPUT A
PROMP "ENTER 'Y' SCALE FACTOR "
INPUT B
PROMP "ENTER 'Z' SCALE FACTOR "
INPUT C
PROMP "ENTER 'X' TRANSLATE FACTOR "
INPUT D
PROMP "ENTER 'Y' TRANSLATE FACTOR "
INPUT E
PROMP "ENTER 'Z' TRANSLATE FACTOR "
INPUT F
OPENO . TRASCL
N=1
ZLOOP GETPOI #A, N, X, Y, Z, K
IF K NE -1, SKIP ZNOSET
K=0
O=-1
ZNOSET X=X+A+D
Y=Y+B+E
Z=Z+C+F
PUTPOI X, Y, Z, K
N=N+1
IF O NE -1, SKIP ZLOOP
PUTPOI 0, 0, 0, -1
DELETE #N
RENAME TRASCL, #A
PROMP "OUTPUT PICTURE ",#A,", NOW HAS ",N," VECTORS"
DELETE TRASCL
*
WELCOME TO PDP-11 GRASS

GETC WITCH

CALL TRASCL

TRASCL -- SOFTWARE SCALE/TRANSLATE UTILITY

ENTER THE PICTURE TO BE WORKED ON? WITCH

ENTER 'X' SCALE FACTOR 72

ENTER 'Y' SCALE FACTOR 72

ENTER 'Z' SCALE FACTOR 73

ENTER 'X' TRANSLATION FACTOR 0

ENTER 'Y' TRANSLATION FACTOR 0

ENTER 'Z' TRANSLATION FACTOR 0

OUTPUT PICTURE WITCH NOW HAS 175 VECTORS

DIRC

CORE DIRECTORY

PICTURE SIZE FLATS

WITCH 608 6X

608 WORDS USED IN 1. PICTURES
10752 WORDS FREE IN 4. FRAGMENTS
This program demonstrates some of the numeric control and dynamic creation commands.

This program inverts any or all of the three axes of a file so an artist who has symmetric data need only digitize one half, one quarter or one eighth of his data depending on how many axes are reflected.

Prompt "REFLECTION UTILITY PROGRAM"
Prompt "ENTER THE NAME OF THE PICTURE ";
Input $A
Getpoi $A,1,F,F,F,F
Prompt "ANSWER THE FOLLOWING, 1=YES, 0=NO"
Prompt "REFLECT 'X' POINTS ";
Input $A
Prompt "REFLECT 'Y' POINTS ";
Input $B
Prompt "REFLECT 'Z' POINTS ";
Input $C
Open $ REFTMP
R=-1
N=1
L00P1 Getpoi $A,N,X,Y,Z,K
If K NE -1,SKIP %NOSET
K=0
O=-1
%NOSET Putpoi X,Y,Z,K
N=N+1
If O NE -1,SKIP %LOOP1
Ren $NOW THE INVERTED COPY
O=0
N=1
%LOOP2 Getpoi $A,N,X,Y,Z,K
If K NE -1,SKIP %NOSET2
K=0
O=-1
%NOSET2 If A GT 0, X=X*R
If B GT 0, Y=Y*R
If C GT 0, Z=Z*R
Putpoi X,Y,Z,K
N=N+1
If O NE -1,SKIP %LOOP2
Putpoi 0,0,0,-1
Delete $A
Ren $REFTMP,$A
Prompt $A," NOW HAS ",N+2," VECTORS"
Delete Reflec
CLEAR
REH *

(IDENTIFICATION GRSUTL-105-V01)
C. D. MOERSDORF

THE ON-LINE EDITOR
*
PROMP "EDIT V1"
XEDIT PROMP "DISK FILE NAME "
REH /I EQ THE INSERT INCREMENT/
REH /AND DEFAULTS TO ONE ON STARTUP/
I=1
REH /L EQ THE LAST LINE <POINTER LOCATION>
L=0
INPUT "A"
PROMP "EXTENSION ";
INPUT "D"
IF #D EQ 'MAC', SKIP XS1
IF #D EQ 'MCS', SKIP XS2
IF #D EQ 'PAL', SKIP XS3
IF #D EQ 'DEC', SKIP XS4
PROMP "??CANNOT EDIT THAT FILE TYPE??"
SKIP XEDIT
XS1 OPENED "A.MAC"
T=1
SKIP XEDIT
XS2 OPENED "A.MCS"
T=3
SKIP XEDIT
XS3 OPENED "A.PAL"
T=5
SKIP XEDIT
XS4 OPENED "A.DEC"
T=7
XEDIT INPUT "E", N
IF #B EQ 'P', SKIP XPRINT
IF #B EQ 'PR1', SKIP XPRINT
IF #B EQ 'D', SKIP XDELETE
IF #B EQ 'DEL*', SKIP XDELETE
IF #B EQ 'I', SKIP XINSERT
IF #B EQ 'INS*', SKIP XINSERT
IF #B EQ 'E', SKIP XEXIT
IF #B EQ 'EXI*', SKIP XEDIT
IF #B EQ 'S', SKIP XSET
IF #B EQ 'SET*', SKIP XSET
IF #B EQ 'R', SKIP XRESET
IF #B EQ 'RES*', SKIP XRESET
IF #B EQ 'L', SKIP XLIST
IF #B EQ 'LIST*', SKIP XLIST
IF #B EQ 'T', SKIP XTYPE
IF #B EQ 'TY*', SKIP XTYPE
IF #B EQ 'Z', SKIP XZAP
IF #B EQ 'ZAP*', SKIP XZAP
PROMP "??UNKNOWN EDITOR COMMAND??"
SKIP XEDIT
XONLINE PROMP "??LINE SPECIFIED DOES NOT EXIST??"
SKIP XEDIT
REH /NEXT ARE THE INDIVIDUAL COMMAND PROCESSORS -FIRST IS PRINT COMMAND-
XPRINT IF N EQ 0, N=L
GETLIN N,N
IF N LE 0, SKIP XONLINE
PROMP N," " ; ; ;
```
SKIP XEDIT
REM /TYPE PROCESSOR/
XTYPE IF N EQ 0, N=L
PROMP "STOP LINE ";
INPUT S
XTYPE1 GETLIN N,K
IF K EQ -1, SKIP XEDIT
IF K EQ 0, SKIP XNOLINE
PROMP K, " *", 
N=-2
IF K LT S, SKIP XTYPE1
SKIP XEDIT
REM /LIST PROCESSOR/
XLIST N=-1
XLIST0 GETLIN N,K
IF K EQ -1, SKIP XLIST99
PROMP/L K, " *", 
N=-2
SKIP XLIST0
XLIST99 FORMP/L
PROMP/L
PROMP/L
SKIP XEDIT
REM /DELETE PROCESSOR/
DELETE IF N EQ 0, N=L
DELLIN N,D
IF D GT 0, SKIP XEDIT
SKIP XNOLINE
REM /SET INCRE PROCESSOR/
XSET IF N LE 0, SKIP XARGER
I=N
SKIP XEDIT
REM /RESET PROCESSOR/
XRESET IF N LE 0, SKIP XARGER
RESET N
SKIP XEDIT
REM /INSERT PROC/
XINSERT IF N EQ 0, N=L
IF N LE 0, SKIP XARGER
XINSERT1 FORMP N, " ";
GETDUF K
IF K EQ -1, SKIP XEDIT
INSLIN N,K
L=N
N=N+1
IF K EQ 0, SKIP XINSERT1
IF K EQ -2, SKIP XNOCOR
PROMP "??LINE ALREADY EXISTS -DELETE FIRST-??*
SKIP XEDIT
XNOCOR PROMP "??NO MORE ROOM TO ADD LINES??*
SKIP XEDIT
XARGER PROMP "??NEGATIVE ARGUMENTS NOT ALLOWED??"
SKIP XEDIT
REM /ZAP (DELETE SEQUENCE OF NUMBERS) PROCESSOR/
ZAP PROMP "STOP LINE ";
INPUT S
D=N
ZAP1 DEELLIN N,K
IF K LE 0, SKIP XNOLINE
N=-2
IF K LE S, SKIP ZAP1
L=D
SKIP XEDIT
REM /EXIT PROC/
EXIT SKIP T
DE'ETE/D $A.NAC
```
SKIP XSTORE
DELETE/D $A.NCS
SKIP XSTORE
DELETE/D $A.FAL
SKIP XSTORE
DELETE/D $A.DEC
XSTORE SKIP T
CLOSEDT $A.NAC
SKIP XEDITOUT
CLOSEDT $A.NCS
SKIP XEDITOUT
CLOSEDT $A.FAL
SKIP XEDITOUT
CLOSEDT $A.DEC
XEDITOUT PROMP "FILE ", $A. "UPDATED AND CLOSED"
IF N EQ 0, SKIP XEDIT
EXIT