THE GRAFFITI GRAPHICS SYSTEM

by

Zdravko Podolski

A thesis submitted for the Degree of Master of Science

Department of Computational Science

University of St. Andrews

1982

ABSTRACT

This thesis describes the design and implementation of the Graffiti Graphics System. Graffiti is a simple yet powerful graphical output system which allows the user to construct pictures as line structures in three dimensions. The pictures may be combined with each other and transformations may be applied to them. In Graffiti, pictures and their attributes are completely separate concepts. Properties such as the type of line, intensity, colour etc, are all treated as different ways of displaying the same conceptual structure. Thus the same picture may be displayed in a variety of ways. This conceptual separation of pictures and modes of display allows us to treat concepts such as perspective display and hidden line removal in a fashion similar to the more conventional attributes.

The graphics features are built as a set of orthogonal extensions to a general purpose high level programming language. The language chosen is S-algol, designed and built at St. Andrews University. The extensions fall conveniently into two categories. Firstly there is a new data type **picture**, together with a collection of operators which perform the basic operations such as transformations and combining pictures together. Secondly, a set of routines is provided for the user to control the display of pictures and for picture construction.

The thesis consists of chapters describing the background, design and implementation of Graffiti, followed by a conclusion and a number of appendices. A survey of existing work is presented and the need for 'yet another' graphics package demonstrated. The conclusion examines the design decisions in retrospect and indicates areas of possible further development. A list of relevant references to published work in the field is included. Appendices are provided, describing the extensions to S-algol, data structures used and devices supported. A user manual, describing in detail the facilities offered by Graffiti, as well as the method of use of the system is also provided.

DECLARATIONS

I declare that this thesis has been composed by myself and that the work it describes has been done by myself. The work has not been submitted in any previous application for a higher degree. The research has been performed since my admission as a research student on 1st. October 1977 for the degree of Master of Science.

Zdravko Podolski

I hereby declare that the conditions of the Ordinance and Regulations for the Degree of master of Science (M. Sc.) at the University of St. Andrews have been fulfilled by the candidate, Zdravko Podolski.

Dr. Ronald Morrison

ACKNOWLEDGEMENTS

I would like to express my thanks to Professor A. J. Cole at St. Andrews University for his encouragement and the use of the computing facilities in the Computational Science Department and also to St. Andrews University who awarded me a Research Scholarship to study for this Degree.

Special thanks are due to my supervisor, Dr R. Morrison for his advice and (ever so gentle) bullying, without whom this thesis would never have been written, and M. Weatherill for the original inspiration.

I must also express gratitude to Professor D. C. Gilles, for the use of computing facilities in the Computing Science Department at Glasgow University.

Last but not least, I am greatly indebted to my wife for supporting me and putting up with me during the writing of this thesis.

CONTENTS

Figures and Diagrams

1. Introduction

1.1. History

The project has its beginnings in a Senior Honours Project in the Department of Computational Science at St. Andrews University in the year 1976/77. The goal there was to build a graphics system on top of Algol-W[1], a language in wide use in St. Andrews at the time. The Senior Honours project succeeded fairly well and produced a library of procedures which could be used to draw pictures on a drum plotter or on the line printer. The picture construction facilities were based on those described by Smith[2].

The Graffiti System was designed and (almost) implemented as a set of extensions to the programming language Algol-R[3]. This language was constantly undergoing change and was eventually scrapped in favour of S-algol[4]. Both the languages were designed and implemented at St. Andrews and although their constantly changing details were awkward the trouble was to some extent alleviated by the fact that their designer was also my supervisor.

The original implementation of Algol-R was for the IBM 360, but was later moved onto a DEC PDP11⊕ running the UNIX* operating system . UNIX is described in [5] and [6] .

S-algol started out on the PDP11 and was then implemented on the Digital Equipment Corporation VAX11 computer and the Zilog Z80

⊕DEC and PDP are trademarks of Digital Equipment Corporation.
*UNIX is a Trademark of Bell Laboratories.

microcomputer. Graffiti should in principle be able to follow S-algol wherever it goes.

The latest Graffiti implementation runs on the PDP11. The host language is S-algol and there are plans to port the system to the Z80 and the VAX.

1.2. What is Graffiti

Graffiti is a simple but powerful general purpose graphics package. The principal aim was to design and build a tool for the construction of specialised graphics systems. Graffiti offers facilities for the description of three dimensional line structures and for their display. Coupled with the generality and power of S-algol data and control structures, the system offers wide opportunities for the budding graphics system designer. Since it is a line drawing package, albeit in three dimensions, there are natural limitations to the system. On the other hand, line drawings have always been one of the most common, effective and widely understood means of communication[7].

In Graffiti pictures are first described then displayed. The most basic form of non-empty picture is a single point. More complex pictures may be formed by connecting two simpler ones together, in which case a line is created between the last point of one picture and the first point of the other. It is also possible to combine two pictures together to form a third, an operation analogous to connection but without the connecting line. General transformations are possible. A 4x4 matrix describes the desired transformation which must be reversible, i.e. the matrix must be non-singular. S-algol syntax is

extended to deal with the picture data type and provide infix operators for connection, combination and transformation of pictures.

A picture is simply an ordered set of points, some of which may be connected. Before such a picture can be displayed it is also necessary to decide on the way the display is to take place. In conventional graphics packages information on line styles, colour, etc, is normally part of the picture itself. These `picture attributes´ are treated differently in Graffiti. They are assumed to be attributes of the display process and not of the picture. Thus it is possible to display the same picture in a variety of styles. More importantly other display attributes or **modes of display**, also respond well to this treatment. Should a picture be drawn using a perspective transform, or is it a purely planar view? Do we clear the screen, eject the paper or should the new frame be superimposed on top of the previous one? All these are part of the display process and need not clutter up the data structure actually describing the line structure that is the picture in question. If a colour graphics device is available, such attributes as colour are naturally dealt with in this way. If the particular display is able to erase previously drawn lines, that too is a property not of the picture but of the way it is displayed. In general, any property which does not materially change the line structure of a picture is not a property of the picture but of the method of display. In particular, removing hidden lines can be treated in this manner and is one of the areas for further development.

A short presentation was given at the 1978 DECUS UK conference describing Graffiti[8].

1.3. Structure of the Thesis

There are seven chapters and five appendices. This introduction is Chapter 1. It is followed by a chapter describing the background in detail, incorporating a survey of relevant previous work, as well as justifying the need for a simple to use line drawing package.

Chapter 3 sets down the design objectives. It thoroughly describes what is going to be done and why. The novel features of Graffiti are explained and the extensions to S-algol are functionally described.

Chapters 4, 5 and 6 describe the current Graffiti implementation in detail. Chapter 4 gives an overview of the implementation plan and discusses its objectives. The data structures used for in core picture storage are explained and the various implementation stages described. Chapter 5 deals with the extended run time support needed to build and display pictures, while chapter 6 concerns itself with the extensions and alterations to the S-algol compiler.

The conclusions are drawn in chapter 7. The chapter also contains a retrospective view of the project.

The appendices contain a list of syntax and type matching rule extensions, a full description of the data structures used to represent pictures, a summary of changes to the compiler and interpreter programs and a User Manual. The User Manual would be all a user needs to write Graffiti programs, assuming knowledge of S-algol programming.

2. Background

2.1. Beginning

The project is a natural continuation of the Senior Honours project in which I took part during 1976/77[9]. That project and my subsequent work were conceived in part as attempts to expand and build upon the ideas described by Smith[2]. In that paper an extension to PL/1 was described, the aim of which was to provide general line drawing facilities. Of the three methods possible when implementing graphics facilities, Smith[2] argues that extending an ex'sting language is the best when it is a general purpose graphics system that is being developed. The other methods are either building a special purpose language, or providing a subroutine library callable from some `normal' programming language. The arguments for and against the three methods are presented in a subsequent chapter. The justification is to take advantage of the base language facilities, avoid reinventing the wheel and cut implementation time. Training time for programmers who would later use the system should also be cut because they are already familiar with the base language and will only need training in the extra features. The difficulty lies in fitting the graphics concepts neatly and orthogonally into the language. As the language is PL/1 in this case, completely lacking in neatness and orthogonality, this becomes a very interesting problem indeed.

In GPL/1 one of the most important considerations is static device independence. The actual output device for the picture is decided not at coding time, or compile time, but at the time the program is loaded,

ready to run. Thus the choice of output device is immaterial when programming considerations are made.

The main extensions to PL/1 are:

1. - vector variables and operations; eg.

   (3X+7Y) + (4X+5Z) = 7X+7Y+5Z

2. - images and image expressions;

Images form the foundation of the system. The image is described as a combination of pictorial data, pictorial function values and other images. Images can be thought of as ordered sets of points, some of which are connected to their successors by a directed line, together with a set of attributes such as colour, intensity, etc. The data structure which is the image reflects the operations which have been applied in its construction. The image operators are:

inclusion

connection

positioning

scaling

rotation

The **connection** operator is similar to **inclusion** , but in addition a line is drawn from the last point of the first picture to the first point of the second picture. The other operators are what one would expect.

An image once created can be used as a building block for other images and the same copy is used for all the images of which it is a part. The decision of the implementor was that should an image be changed, all the images of which it is a part will also change. This can however produce confusing side-effects. Due to mainly financial constraints on memory size in computer systems this approach could be considered justified. These constraints are rapidly disappearing.

Thus two different data types were added to PL/1, vectors and images. Vectors correspond to points while images are scenes built up of points, other images and attributes such as colour, intensity, etc. Vector arithmetic is available which makes life much easier for the programmer.

The ideas and concepts in GPL/1 were novel and attractive and I embarked upon the design and construction of a graphics system along similar lines.

2.2. Other Published Work

In order to work confidently in the knowledge that others have not trodden the same path and to gain insight into the problems and intricacies of graphics systems in general, it was necessary to undertake a survey of literature on computer graphics. The aim was to learn what a graphics system really is, in other words to understand the model a programmer might have in mind when using such a system. One of the first papers studied, about the proposed CORE graphics standard[10], states this beautifully:

> "It is very helpful, in designing any kind of computer system, to try to provide the system user with a simple, coherent **model** of the system. Without this model, the user will find it very difficult to make proper use of the system. We consider this an important concept for the designer of a standard graphics package. The programmer must be given a simple set of functions, built upon an equally simple set of abstract ideas."

The CORE proposal in fact presents not just a design specification, but also a whole methodology for the design of a standard. It also attempts to explain the need for such a standard and sets out the objectives to be achieved by it. Portability, both of software and programmers, ranks near the top of those objectives. The philosophy of graphics systems is examined in detail and a full functional specification of the proposed standard is given. The final section is extremely useful to any aspiring graphics system builder as it examines in detail the issues leading to the functional specification.

The benefits of standards are unquestionable, but CORE attempts to be all things to all people. It is also a subroutine based system, rather than one using abstract data types and operators upon them like GPL/1 and like Graffiti. Nevertheless, this is a most valuable paper.

In the same issue of Computer Graphics is a survey[11] of several of the best known graphics systems then available. This paper surveys the technical aspects of software for line drawing displays, with a view to providing a framework for the comparison of graphics software

packages for line drawing graphics systems. The terms of reference are somewhat restricted, especially as the packages were only considered if they could be used with a Fortran application program. The paper compares ADAGE, CalComp, DISSPLA, GCS, GINO, GPGS, IG and Tektronix packages and includes a useful table of features. It also contains a list of references to the packages reviewed.

Due to the enormous mass of literature about computer graphics some criteria had to be used to select the articles to read. The two main criteria were that a paper had something to do with the treatment of pictures as data objects, or had something to offer about transformations of pictures. Nevertheless, some references are given for graphics packages of particular relevance, or standardisation efforts, or descriptions of particular techniques. In particular, several surveys were studied, the most complete being two in Computing Surveys by Williams[12] and Freeman[7]. Williams deals with data structures and Freeman with image processing and both include substantial lists of useful references to other works of relevance to a designer of a graphics system for the construction of pictures. The emphasis of most papers seemed to be on image processing, pattern recognition, etc, which is in fact the analysis of pictures. The articles on purely descriptive systems were fewer in number, and amongst them many were concerned with a particular application, such as for example circuit design.

Graffiti objectives and features are outlined in the previously mentioned paper in the proceedings of DECUS UK 1978[8].

In Graffiti we are concerned with three dimensional **line structures.** In this day and age one might feel that this requires some justification. Freeman[7] provides it amply and I paraphrase:

A line drawing is one of man´s most effective means of communication. Road maps, weather maps and engineering plans are all line drawings. Printed text itself is a pattern of lines. Not only are there many graphical presentations directly recognisable as line drawings, but photographs etc, can also be interpreted as contour maps, say, without appreciable loss of information. Nevertheless, it must be admitted that line drawing has, like any other technique, its limitations. Smooth shading of surfaces is totally impossible for example. Representing curved surfaces can only be done by drawing several lines together, which is an unsatisfactory technique when realistic scenes are needed. However, the sort of output device which Graffiti is designed to run on is most unlikely to have the resolution necessary to display realistic scenes anyway. Graffiti is supposed to be primarily simple and easy to use, designed for inexpensive computer systems, coupled with cheap and simple display devices.

Regarding published works about graphics, one cannot but mention the seminal paper about Sketchpad[13]. Much more involved than this attempt, Sketchpad nevertheless introduced the idea of ring structures combining pictures together, and enabling easy interaction between humans and a computer program. Such rings could be implemented in Graffiti too, although it was not designed for graphical input, using the data structuring capabilities of the high level language in which it is embedded.

For a tremendous amount of background information, and detailed advice on interactive graphics, the book **Principles of Interactive Computer Graphics** by Newman and Sproull[14], is required reading.

Although Newman and Sproull contains adequate mathematics, **Mathematical Elements for Computer Graphics** by Rogers and Adams[15], may be better suited for just looking up methods of inverting matrices and similar necessary, but definitely not interesting, details.

The GPL/1 paper[2] has already been mentioned as the basis for the Graffiti work. There are however several other papers treating pictures as objects on which operations are performed. Childs[16] describes a machine independent data structure, which is designed to be of a general nature. The approach is set theoretic, and is based on a small number of **generator sets.** These are combined together to form **composite sets.** Only the generator sets have storage representations and composite sets are generally collections of pointers to and relations between the generators. Composite sets can be thought of as unions of generator sets and the generator sets themselves are mutually disjoint. Graffiti can be thought of in terms of a single generator set, consisting of the **point**, together with a relation **connects to** between two points, which can be true or false depending on whether a line exists between the two points or not. Graffiti imposes a little more structure on pictures, by insisting that points be ordered, and lines directed.

Of interest is work by Bracchi and Somalvico[17], which, although concerned only with circuit design, does contain some advanced features. An important concept is that of **circuit variables.** These can be single nodes, for example a transistor, or complicated circuits. The nodes can

be thought of as the generator sets, whereas the composite sets are produced by the operations of connecting (restricted to nodes) and inclusion (denoted by the symbol +). The circuit variables can also be scaled, rotated and moved. The system is restricted to two dimensions.

A subsequent paper by Bracchi and Ferrari[18] describes a more general system, still in two dimensions. This is implemented as a set of Fortran extensions. Two new data types were added, that of **graphic** and **geometric** variables. A geometric variable would not have an image, but would be used in the construction of graphic variables. Examples of geometric variables are arcs, circles, lines, squares and the like, while graphic variables are one or more planes of geometric expressions. Circuit masking and similar applications were facilitated by allowing operations on graphic variables. These are **intersection** , **union** and **difference** with the appropriate set-theoretic meanings. Patterns can be stored as functions, and then repeatedly invoked.

Jones[19] describes an interesting graphics system where pictures are treated as macros and transformations can be applied to pictures or parts of pictures. The transformations are **scale, rotate** and **move.** However one of the most impressive systems is described by Greenberg[20]. This system runs on a dedicated configuration and is concerned primarily with enabling easy transformations of pictorial data. Techniques are described which achieve the separation of points data and transformation data, resulting in space and time saving algorithms. Standard procedures are provided to the users for input and editing of graphical information and extremely promising results have been achieved. The paper includes some beautiful pictures.

The linguistic approach to picture analysis and less commonly synthesis also appears very interesting. An early survey of such work was done by Miller and Shaw[21]. The paper by Stanton[22] is also most instructive. In general the emphasis here is that of analysing a picture using a syntax analyser of some description. The application to generation of scenes is in constructing a few basic elements, corresponding to the generator sets described by Childs[16]. The basic elements can then be combined using operator expressions to produce pictures. The treatment of pictures is uniform, since a complex picture can be used in the place of a simple object in an expression. There is some similarity between this and the Graffiti approach.

Another paper of relevance is by Warner[23]. Here a graphic composition is defined as any spatial model formed from basic dimensional **primitives** such as points, lines and characters. An **object** is a named set of contiguous dimensional primitives. **Compositions** can then be developed from instances of these objects and complex compositions are built up from groups of object instances, etc. Thus we get a compositional hierarchy. The units of the environment are determined by the composition designer and the system offers easy prototype object definition capabilities and the ability to define multiple primitive instances and hierarchical groups. Transformations are simple and flexible display techniques are provided. The system, called MIDAS, is designed principally with compositional modelling in mind, but the principles involved could be generalised.

An interesting method of describing three dimensional bodies is described by Ricci[24]. The objects are described as unions of

functions of x,y and z. The value of a function at a point is less than 1 if the point is inside, greater than 1 if the point is outside and exactly equal to 1 in the point is situated on the boundary. Suitable combinations of solids are constructed by unions and intersections of the functions.

For those interested in the progress of various standardisation efforts, Computer Graphics (SIGGRAPH-ACM) publishes regularly lists of implementations of the CORE[10] system, in particular volume 13 no 4. Papers about proposed standards different to CORE are [25] and [26]. A survey of various standardisation attempts has been undertaken by Newman[27].

A few recent publications of some relevance to Graffiti are listed below.

Kilgour[28] presents an outline model for a hierarchical system, with modules at various levels talking to each other. Commands travel down the levels and event records are returned upwards. The model is more general than CORE or GKS and some possible applications are described.

The paper by Sint[29] gives a short description of an implementation of ILP[30], an intermediate language for pictures, using a set of extensions to Algol 68R. In ILP a picture consists of various attributes as well as the line structure and transformations on it. The data structures used are directed acyclic graphs and structured data facilities of Algol 68R are used to create what is essentially a new data type.

Keith[31] describes a general transformation structure that will accommodate varying orders and levels of transformations on sets and subsets of points describing objects. The structure is essentially a binary tree and transformations are represented using 4x4 matrices.

Computer Graphics occasionally publishes lists of packages such as [32] and also lists of references to papers, Masters' and Doctors' theses.

To use Graffiti effectively, some familiarity with S-algol is necessary. Cole and Morrison[33] is an excellent introduction.

3. Design

3.1. Introduction

There is a wide variety of graphics software available, for almost any application. However, it seems that most of the existing packages are generally too preoccupied with particular features of the available hardware, or with particular aspects of the problem to be solved. Neatness and generality are often sacrificed for the sake of giving the user access to a particularly obscure feature of some device.

I hope to expose some of the underlying concepts of computer graphics at the base level and the design of Graffiti should allow the programmer to build upon them. The objectives of the project were thus the following:

a. Design a basic graphical output system, which is easy and simple to use as well as not requiring expensive and complex hardware to run on. The components must fit together consistently.

b. The basic facilities must be able to be used as building blocks for the implementation of more complex systems. The user should not be barred from using the full power of whatever machine and operating system he may eventually use. This means that an attempt has to be made to extract the highest common factors from the, often confusing, mass of features and facilities offered by today's graphics systems.

This chapter describes first the general model of a graphical output system, then goes on to explain the design decisions and the

features of Graffiti. The adopted model, following closely upon the general one, is shown and the extensions to S-algol described.

3.2. The General Model

Considering a graphical output system on its own, there is a fair degree of agreement as to the overall structure that is appropriate. Kilgour[28] shows the typical structure to be that of Figure 3.1. The various components are:

a. The application program which creates a **high level picture description** , a view independent data structure which defines the scene to be drawn. The coordinate system is chosen for the convenience of the application program.

b. The viewing transformation transforms and clips the high level description. The output is a **low level picture description** which is two-dimensional but still device independent. The coordinate system is that of an idealised device, referred to in the CORE proposal[10] as **normalised device coordinates .**

c. The device converter which converts the commands of the above low level description into a **device level picture description** directly executable by the relevant device. The description is expressed in the device coordinate system.

Any or all of the picture descriptions may be absent in a particular system. The Graffiti model will be presented after the design decisions and features of the system have been discussed.

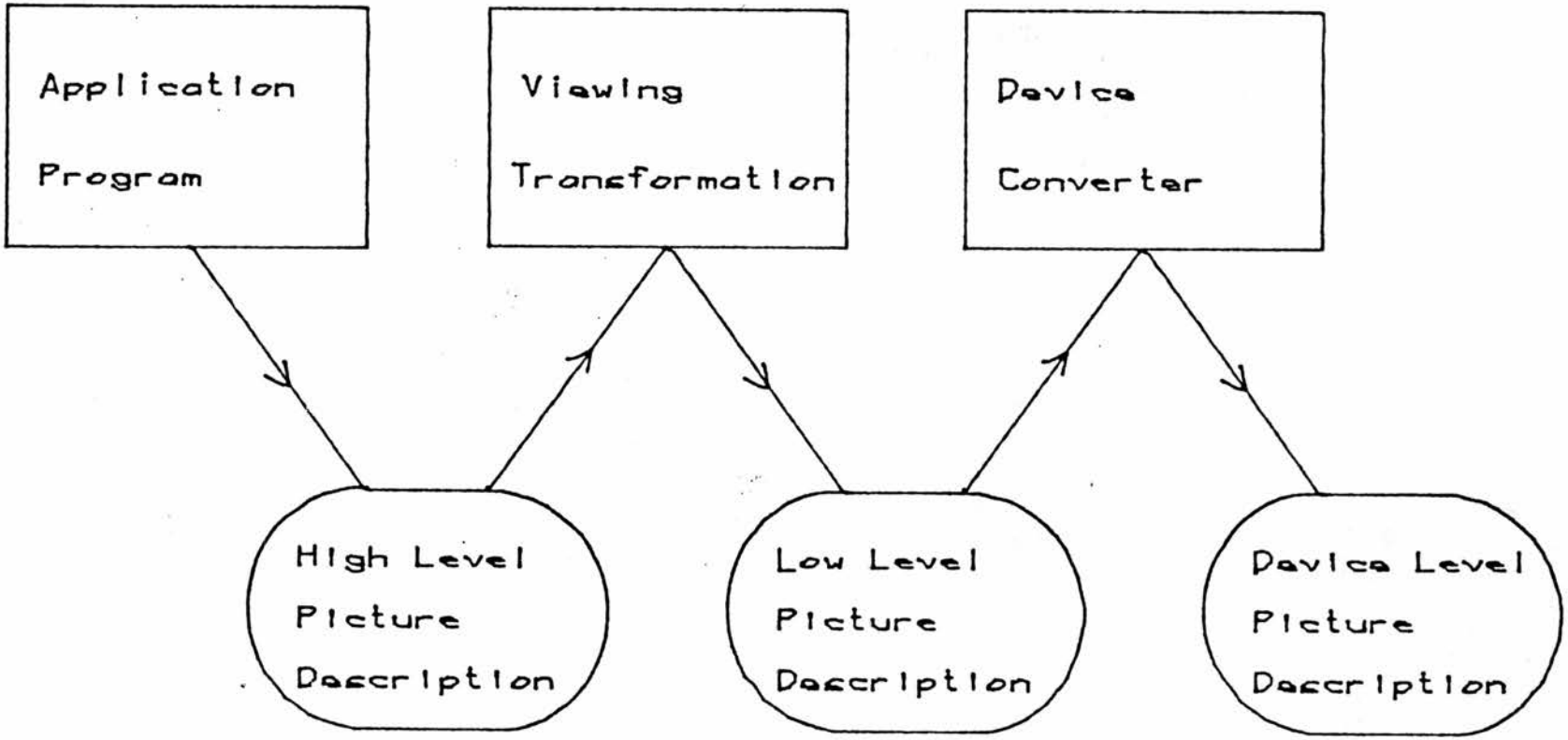


Fig 3.1: Model of a Typical Graphical Output System

3.3. Design Decisions

Smith[2] sets out the three distinct approaches which have traditionally been used for the design and implementation of graphics systems:

1) Design a special purpose graphics language and/or system. Examples are Sketchpad[13] and GPDL[34].

2) Implement a suite of subroutines and programs which may be called from one or more languages. Eg. GHOST[35].

3) Extend an existing programming language to include the required facilities. One example is GPL/1[2]. A more recent example is described by Sint[29].

The first method, designing a special purpose system, is a very attractive solution as it allows the designer complete freedom of invention. Traditionally though this method tended to result in a package which could only be used on one particular hardware configuration. This could possibly be because the effort involved in designing and implementing such a system is relatively significant and the designers might not attempt it unless they had specific hardware available all ready and waiting.

The second method is very popular. Subroutine packages are widely used and with considerable success. The method is certainly easy to implement but suffers either from limited capabilities or cumbersomeness in use. It would be desirable to have a subroutine library available to several languages but in practice it is difficult to standardise

subprogram calling conventions. Subroutine libraries can generally only be used from one programming language.

In any graphics package there is usually some degree of overlap between the techniques. In particular there is usually a suite of subroutines provided with any graphics system.

The technique of extending an existing language has not been all that widely used. This seems to be mainly because the designers and implementors need a familiarity with the compiler for the language which is only achieved by much effort. However if carried out carefully it can give very good results.

Graphics applications of any size and complexity almost invariably require some non-graphics computing support. Any proposed system should at least allow access to a general purpose programming language or be an extension of such a language. Smith[2] lists some of the advantages of the "base" language approach:

A. Provide a syntactic base to guide in the design of the graphics statements.

B. Cut implementation time because less needs to be implemented and because most or all of the implementation can be done in the host language.

C. Take advantage of extensions to and improvements of the base language.

D. Avoid "reinventing the wheel".

E. Cut training time when the base language is one known to the users.

Graffiti is therefore a set of extensions to a general purpose programming language. Algol R[3] was originally chosen for the task but was soon superseded by S-algol[4]. S-algol is a language designed and first implemented in the Computational Science Department at St. Andrews. It features a wealth of data types and control structures well suited to complex problems. The task of gaining familiarity was not too difficult, as the implementors were at hand to give advice and help with the internal workings of the compiler. Needless to say, the project provides a comprehensive test of the S-algol compiler and the run-time facilities, giving increased confidence in their reliability. The compiler is itself written in S-algol and it was hoped this would considerably reduce the difficulties in altering it.

To remain consistent with the philosophy of simplicity and orthogonality the extensions to S-algol have to be as few as possible. The resultant language is a true superset of S-algol, meaning that the Graffiti compiler will compile all valid S-algol programs as well as those using the extensions.

Considerations of finance, time and hardware available combined led to the specification of Graffiti as a graphical output system only. Nevertheless, the S-algol language enables the construction of features for graphical input as well. True graphical interaction is not a part of Graffiti, that is left to the eventual implementor using Graffiti as a building tool. The system supports three dimensional line structures.

The design and programming effort necessary to include the extra dimension is not very great, but mechanisms for projecting three dimensional scenes into two dimensional line drawings had to be provided.

One can easily use a computer to process three dimensional structures, but line drawings can only uniquely render two dimensional line structures. Any projection mechanism must of necessity introduce ambiguities and therefore facilities must be available for viewing the picture from several viewpoints and using more than one method of projection.

Given that we are using line structures, there are three ways to obtain them:

(i) An object can be abstractly specified in a geometric sense, as described by Ricci[24], for example.

(ii) The lines can be traced, as in GHOST[35].

(iii) A model for the information conveyed by an image can be built, eg. GPL/1[2].

In any case, unless the object being represented is composed entirely of lines, all three approaches are only approximations to the real scene. Since most pictures, no matter how perfect, are themselves still only approximate representations of the real scenes this is not felt to be a great drawback. In the case where the picture itself is the real thing, as in diagrams, line drawings are almost invariably used. What matters is how good and useful an approximation we can

achieve. The Graffiti approach is the third one, that of building a model of the real object.

It is also necessary to have some real output device on which any pictures may be drawn. A Tektronix⊕ 4006[36] was purchased by the Department. This is a simple storage tube device, doubling up as an ordinary computer terminal. The instruction language for this device is applicable over most of the Tektronix 4000 range of displays and plotters and is also emulated by many other manufacturers.

3.4. Graffiti Features

Graphics output devices are generally similar to other output devices and should, as far as possible, be treated as such. Output is sent to them, and they perform some action dependent upon that output. Output to, or indeed input from, a graphics device is a stream of binary information. This information is decoded by the device and with luck a picture appears. When sending output to say, a line printer, relevant information has to be included for line feeds, carriage returns, page throws, and so forth. A purpose built graphics device differs only quantitatively from line printers, visual display units, etc. A simple storage tube terminal such as the 4006 can be used as another vdu, whereas devices which cause interrupts, perform direct memory access or whatever, need special handlers in the host operating system. Ideally the end user should not need to worry about such considerations. He should be able to perform input/output operations using normal

⊕ Tektronix is a registered trade mark of Tektronix Inc., Beaverton, Oregon, USA.

subroutine or system calls and the device should perform accordingly. High level languages have long had facilities for transferring information to and from i/o devices. By providing similar facilities for graphics devices we may then hope to write the whole system in a high level language.

Therefore, although restricting ourselves in the first instance to an output system, with an interface only to the Tektronix 4006 and similar terminals, the design decisions were made with a view to enabling the use of Graffiti for other hardware in general and input devices in particular.

3.4.1. The Coordinate Space

One of the very first decisions that had to be made is that of systems of coordinates supported. Higher level graphics systems normally use a right handed coordinate system where the **x** coordinate increases towards the right, the **y** coordinate upwards and the **z** coordinate towards the person viewing. The left handed system differs only in having the **z** coordinate increase away from the observer. The right handed system is usually translated into a left handed system by the viewpoint transformation, before the depth (z) coordinate is stripped off. As Graffiti is a low level system I feel that there is no need for the extra transformation this involves and have decided to use a left handed system for the description of pictures. The user is of course implicitly free to use either system, as the transformation is that of simple reflection. There is no provision for polar coordinate systems, but they can easily be implemented using appropriate

conversions. One possible way to allow different coordinate systems would be to allow the user to specify the system needed as one of the modes of display described below. Therefore, as far as the user is concerned when describing a picture, the axes are as follows:

x- increases towards the right;

y- increases upwards and

z- increases away (into screen, paper or whatever).

There is just the one coordinate space for all pictures, being the equivalent of the mathematical space in GHOST[35]. This space is immutable with respect to the coordinate system. Should the user wish to achieve the GHOST equivalent of a different mapping between the mathematical and physical spaces, this is done by applying transformations to pictures. It has always seemed a little presumptuous that to achieve a simple change in ratios between parts of a picture one must change the universe. Surely it is the picture which should change.

The user describes the pictures in terms of lines in the Graffiti mathematical space and then defines a `viewing box´. Only that part of a picture which is within the viewing box, after perspective transforms if specified, is potentially visible when a picture is displayed. The edges of the box are parallel to the axes and the front face of the box (i.e. the nearer of the two faces parallel to the x-y plane) is mapped onto the output device. The mapping between the `mathematical´ and `physical´ spaces depends therefore on the viewing box. The front face is projected onto the output device so that the ratios of the sides is unchanged and the largest area of the screen is utilised (see fig 3.2).

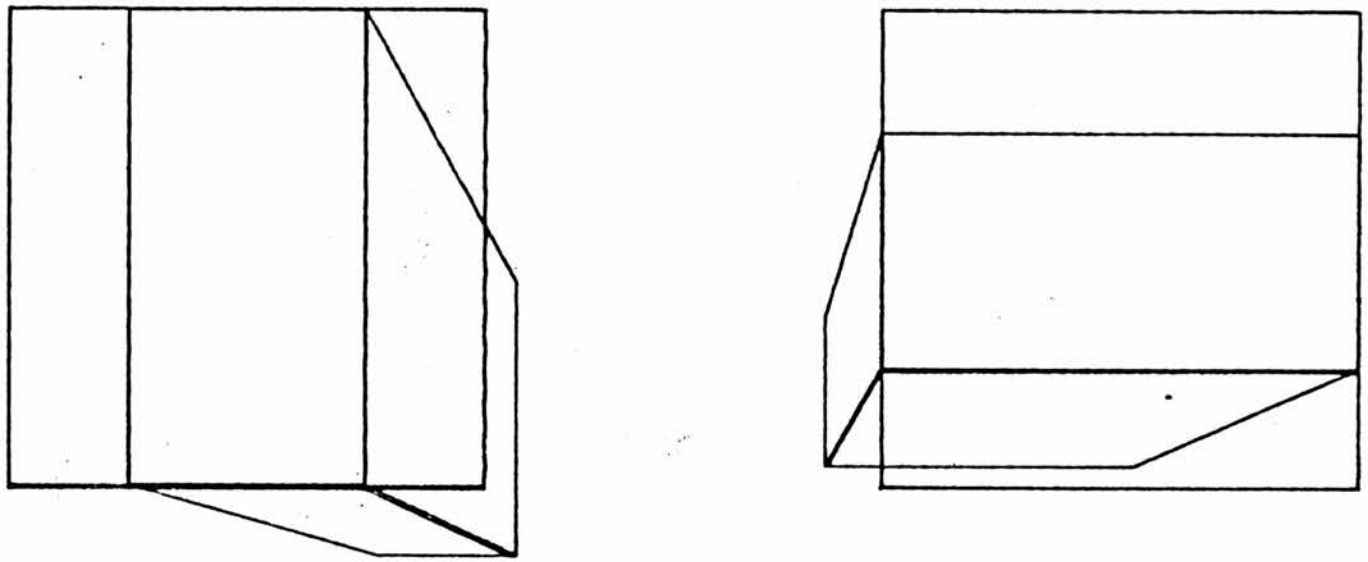


Figure 3.2: Viewing box to device mapping

The faces of the box are treated as being opaque and the user should define a window on the front face through which he will look. This viewing box plus window mechanism allows users to subdivide the screen if necessary, for viewing different pictures, or different aspects of the same picture. Naturally, suitable default values for the box and window are used if the user chooses not to define his own.

For the sake of convenience the eye is considered to be located on the line passing perpendicularly through the centre of the front face of the box. The user is able to specify the distance of the eye from the box to achieve perspective viewing, otherwise the distance can be thought of as infinite and no perspective transformation takes place. Should it be necessary to view from any particular viewpoint, in any direction, the relevant transformation can be simply calculated. This method allows the user full generality of viewing pictures with the minimum of specification. The more the user diverges from the provided defaults, the more detail must be provided to the system. Figure 3.3 shows an example of a viewing box and window. Any part of the picture which lies outside the solid bound by the window size and the depth of the box will be clipped off. Thus the user need not worry about clipping and can make sections through solids simply by defining suitable viewing boxes and windows.

3.4.2. Language Extensions

When designing a system for the description and output of pictures such as Graffiti, one is faced with the choice of sending the graphical information to the output device as it is described, or building up a

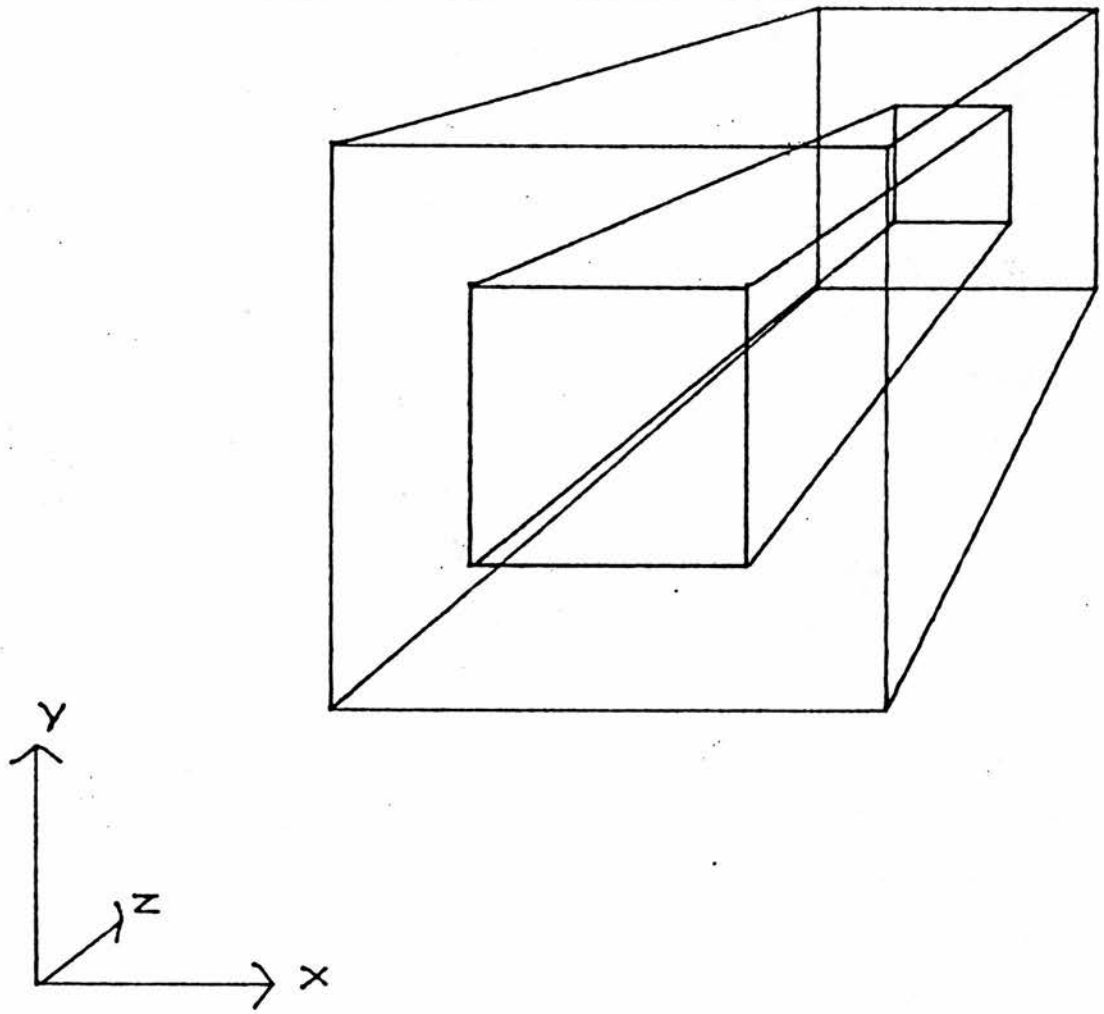


Figure 3.3: Viewing Box with window

data structure in memory (primary or secondary) for later output. In fact choosing the latter alternative leaves them both open, as it allows the user to get arbitrarily close to the former, by describing pictures as simple as necessary and displaying them superimposed upon each other. Also, the majority of computer systems nowadays allow virtual memory space to be much larger than the actual physical store available and thus memory usage is no longer an important consideration. The second alternative was therefore chosen. Another reason is that this enables a neat dichotomy to be achieved between the description of pictures and their output. The extensions to S-algol therefore fall into two categories:

1) A new data type pic and a set of operators for creating and manipulating instances of the new type.

2) Mechanisms for actual display of pictures on some graphics device.

A picture is the value of an object that may be drawn. The data type has all the rights of any other in S-algol, i.e. pictures are first class citizens. There are picture variables and arrays, procedures may return a picture as their value (analogous to Fortran functions) and pictures may form parts of structures. Pictures can also be used together with pictorial operators, in forming picture expressions.

After a picture has been satisfactorily constructed it will have to be displayed. It is necessary to provide routines for the display of pictures and for the setting of display parameters such as perspective, viewpoint, superimpose or start a new frame, etc. These are called **modes of display** in Graffiti and are controlled separately from the

pictures themselves.

3.4.2.1. The Graffiti Model

The general model described above applies closely to Graffiti. The structure is the same, with one extra feature added - the **environment description** (see Fig. 3.4). The environment contains all the information about the modes of display. The limits of the viewing box and of the window, the viewpoint distance, whether the picture is to start a new frame, etc, are all part of the environment in which a particular picture is displayed. The environment, consisting of all the modes of display, is described separately from the picture itself.

All the components of the model are parts of the same program. The alternative is to have several programs, behaving as filters[37], performing the transition from one level to the next. The user writes the Graffiti program and at load time decides which device converter to use. Thus it is simple to provide static device independence.

3.4.2.2. Pictorial Operators

A picture is best described as an ordered set of points in three dimensions, together with a relation **connects-to** which may or may not be true for any pair of adjacent points in it. The concept of ordering of points is due to Smith[2]. All the pictures are considered to exist in the same three dimensional coordinate space. The user supplies the relevant information which, if correct, ensures that the picture can be meaningfully displayed.

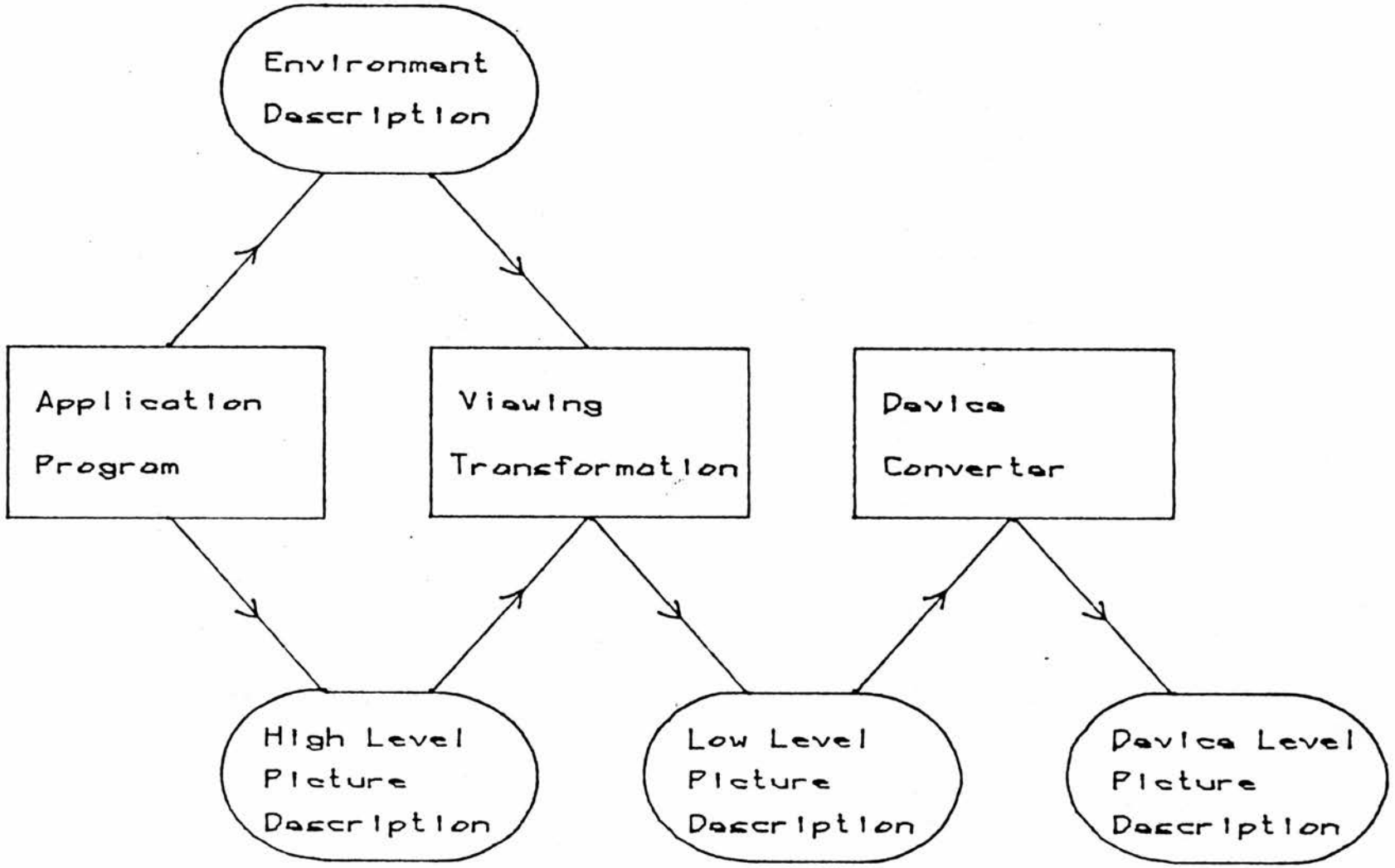


Fig 3.4: The Graffiti Model

Since the points in a picture are ordered, so that there exists a first and a last point, the relation **connects-to** generalises to arbitrarily complex pictures. An operator is provided to achieve connection of two pictures. It is called the **connect** operator and is written ^.

It is also necessary to be able to combine pictures together without extra lines being constructed. The combination of two pictures is defined as being connection, but without the line joining the two pictures together. The order of points within the subpictures is preserved in both cases, with the first point of the second picture coming immediately after the last point of the first picture. The **combine** operator is written **&**.

Before any picture can be meaningful it must contain at least some points. In Graffiti points themselves are never drawn but only lines, so a picture consisting only of points will not be visible. However it is possible for a line to be of zero length and the system will produce a spot at that point. Before any lines can be constructed there must be a way of specifying points. The notation follows closely that of the vector initialisation in S-algol. The three coordinates are written separated by commas and enclosed in square brackets as in **[x, y, z]**.

The majority of graphics systems allow some sort of transformation to be performed upon a picture. The common transformations are translation, scaling and rotation. Sometimes shearing is also available. In Graffiti the only restriction is that a transformation must be linear and reversible. This means that any transformation can be represented by a non-singular 4 by 4 matrix. To apply such a

transformation the **transform** operator, written **#**, is provided.

Therefore a picture in Graffiti may be one of several things. It may be a single point, or one picture connected to another, or a combination of two pictures, or a transformation of another picture. The operators chosen for the creation and building of pictures are closely involved with the ordering of points. The idea of the connection and combination operations was first described by Smith[2]. The ordering is defined as follows:

(i) If a picture consists of a single point that point shall be the first and last point of the picture.

(ii) When a picture is connected to or combined with another one the order in which they are specified determines the ordering of points in the resultant picture. The order within each picture is unchanged and the last point of the first picture immediately precedes the first point of the second picture.

The extensions to S-algol are thus one extra scalar data type, pic, three infix operators and an operator for writing down points.

We list the operators:

1. the **point** operator - creates a picture consisting of the single point: [x, y, z]

   eg. let pict = [0.0, 0.0, 0.0]

2. the **connect** operator - creates a picture containing both operands with a line between them: p1 ^ p2

```
eg. let pict = [ 0, 0, 0 ] ^ [ 1, 1, 1 ]
```

3. the **combine** operator - similar to connect, but no extra line is created: p1 & p2

```
eg. let pict = ([1, 0, 0] ^ [0, 1, 0]) &
               ([0, 0, 1] ^ [0, 0, 0])
```

4. the **transform** operator - takes a picture and a 4 by 4 array of real numbers and produces a suitably transformed picture:

```
pict # matrix
```

eg.

```
let mat = @1 of c*creal [
    @1 of creal [1, 0, 0, 0],
    @1 of creal [0, 1, 0, 0],
    @1 of creal [0, 0, 1, 0],
    @1 of creal [0, 0, 0, 1] ]

let pict1 = pict # mat
```

Pictures are built up by starting with points, connecting them together, combining them and applying transformations to them. Arbitrarily complex line structures can thus be constructed. For example in circuit design one might create pictures called resistor, transistor, and the like. They would then be used to build more complex pictures of gates which could then be used for adders, etc.

Each picture can be defined to be of the same size and in the same position. To include components at the correct places in the circuits matrices representing translation, rotation and scaling would be used. These and other linear transformations in three dimensions can be represented by 4 by 4 matrices. If a point is thought of as a column vector with four entries, then transforming it is just multiplying it on the right by the transformation matrix, as in Fig. 3.5.

$$\begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix} \times \begin{vmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{vmatrix}$$

Fig. 3.5 : Transforming a point

If a matrix should happen to be singular, i.e. does not have an inverse, this can cause problems. The most obvious consequence is that one cannot reconstruct the original picture from the transformed one. Although such transformations can still be meaningful, in Graffiti all transformations must be reversible. A fuller discussion of transformations and the representation of points and pictures can be found in Newman and Sproull[14].

3.4.2.3. Subroutines and Functions

The second set of extensions to the host language consists of additional standard procedures. Provision is made for the display of pictures as well as the manner in which the display is to be performed. I feel that the action of displaying a picture is separate from the picture itself. What we are dealing with are line structures and they consist of lines, not of any extra details such as colour, intensity, etc. The line structure is not green or pink, although its display colour may be. It is not necessary to keep information about how to display a picture with its line structure. I feel it is also not desirable because we are trying to work with the most primitive concepts in graphics. Picture attributes should not be part of the bottom level structure. Factors such as colour, intensity, whether the lines are dashed or unbroken, should this picture be superimposed onto a screen or paper already containing an image or should the screen be cleared, paper ejected or whatever, are nothing at all to do with the picture itself. A square remains a square, be it pink or purple, whether it blinks or is drawn using broken lines. Properties such as these are functions of the displaying action. Some of them may depend on a particular piece of hardware, as for example blinking, others may be more general. For example starting a fresh screen has its analogues in all graphics devices, even in such exotic media as oil paintings and holograms.

This method of treating all attributes as modes of display extends easily to perspective and viewing transformations in general. Hidden line removal could be just a mode of display like any other.

At the present time hardware is changing extremely rapidly. Until recently colour displays were extremely expensive, whereas they can now be obtained for a modest outlay within the reach of a dedicated personal computer buff. This trend can only be expected to continue and a graphics package purporting to strike at the roots of computer graphics must be able to cope. Separating pictures and modes of display allows flexibility in this respect. Intensity variation is currently not available on cheaper devices, nor are hardware transformations, but they may very well come along in the next few years. The user must be able to obtain some meaningful display of a picture whether the particular device actually supports the given feature or not. Thus transformations and hidden line removal may be done in software, colour and intensity ignored and default actions provided for all modes of display.

The modes of display should therefore be implemented separately from the pictorial data structures. They should not preclude display taking place even if the particular hardware does not support them. This leads us to give them default interpretations. These default actions are taken if the hardware is not able to perform the desired function and also if the user does not specify how the picture is to be displayed. Colour, for example, would be ignored on a monochrome terminal. User knowledge of a specific device need not extend further than knowing which Graffiti features are supported for that device.

Because S-algol contains all the normal data structuring and control capabilities of a modern high level language the user is not precluded from building `super pictures´ which can contain the display information as well as the line structure. In fact this may be

precisely how a higher level system may be constructed. Graffiti provides the most general building blocks, so that sophisticated applications may be designed and implemented with relative ease. Ring structures, first described in the Sketchpad report[13], and later used as a basis for other systems can very easily be implemented using the structure concept of S-algol. Graffiti is more general than either a special purpose graphics language or a suite of subroutines as described earlier. Of course, both these approaches may be implemented using the Graffiti facilities.

Obviously the most needed procedure is one to plot a picture. Others are provided to alter the modes of display and include at least provision for defining the limits of the viewing box and the window, viewing distance, whether the picture starts a new frame or not and whether frame and window borders are drawn. A full external specification of each procedure is given in Appendix D, the User Manual.

3.4.3. Miscellaneous Facilities

It is desirable to provide a few extra facilities in Graffiti which are not strictly necessary, but which can make life a lot easier for the user. The main such feature is provision of facilities for inclusion of text in pictures. It is possible, where the display can act as a normal output device, for the user to use the S-algol character output facilities. Alternatively the user could easily define a character set in terms of lines, but this would be tedious and complicated. But since most graphics output devices are able to display textual information Graffiti allows the user to place a character string at a given position

using a procedure text which takes a string and three coordinates and returns a picture containing the text starting at that point. The text will be displayed in a device dependent manner when the picture is plotted.

To ease transformations, procedures are provided to construct appropriate matrices for translation, scaling and rotation. The user simply supplies three real numbers which specify the amount of transformation in each of the axes and the matrix is returned ready for use with the # operator.

Procedures for saving pictures in files and incorporating such pictures in new ones are provided too, as is a procedure for constructing a picture of a dot or polygon centred at a given point. Also available is a procedure for clearing the frame. These facilities may be expanded in the light of experience.

4. Implementation Plan

4.1. Introduction

This chapter deals with the implementation plan for Graffiti on the PDP11/45 computer running the UNIX operating system.

A discussion of the objectives is followed by a description of the various stages of implementation and testing.

The decisions taken are presented and explained and the data structures used in the representation of pictures are described in detail with illustrative examples.

4.2. Implementation Objectives

The first and foremost objective of the Graffiti project was to produce a simple and easy to use graphical output system. The design stages were oriented towards simplicity and orthogonality and the implementation should not destroy these properties. Portability of application programs is the second goal and one which depends greatly on the implementation. Thus S-algol was chosen as the implementation language, being simple and easy to learn, as well as being the base language of the Graffiti system itself. The language is interpreted in all its current implementations. There are only two things needing done to transport the language to a new system, implementation of a new interpreter and the modifications to the run time support enabling programs to call on the operating system for services. The interpreter itself is designed using stepwise refinement methods and is thus easily

rewritten. The run time support is in two parts, some of it is in the interpreter, some in precompiled form to be bound with the user programs. The routines for precompilation are all written in S-algol anyway and are thus easy to change.

In principle, therefore, the S-algol compiler can be moved to a new system with the minimum of fuss. Therefore Graffiti will be as portable as S-algol. In practice it is likely that minor changes will be necessary to the compiler source itself, as different operating systems use different conventions about input and output as well as possibly having storage locations of different sizes from the original system. Also the device dependent parts of Graffiti are by their very nature not portable, unless the target system has similar devices available.

There are three parts to Graffiti as far as the implementation is concerned. Firstly the compiler itself, which is an extended version of the S-algol one. Secondly the interpreter which is in fact the S-algol interpreter. This minimises the portability problems, as the interpreter is usually written in machine code for speed of execution and is therefore the most difficult part to port to a new system. Thirdly there is the run time system. This incorporates the user level routines such as plot, as well as the code necessary to effect the desired transformations, construct pictures, etc. The run time portion of Graffiti is itself written in S-algol and contains only a minimum of operating system dependencies. These are all well signposted and are mostly concerned with the input and output. The routines dealing with the transformation to device level picture description and the subsequent output of commands for a particular device are inherently

dependent on the device being used. However they should be usable on any operating system that actually has such a device. The conversion to device level description is the very last action Graffiti takes when a picture is plotted. Thus it is well isolated within the system and it is possible to build up libraries of device converters which can the be loaded with the rest of the run time system to achieve static device independence.

Thus to minimise portability problems, only the S-algol compiler was changed. The interpreter is the same. The run time system should run on any system supporting S-algol thus achieving for Graffiti the same portability as S-algol.

4.3. Implementation Stages

Before implementation commenced it was necessary to decide on the way in which it was to take place. The stages should be as far as possible self contained and programs must be able to be tested before the whole system is complete. Thus confidence can be built up and newly introduced errors easily identified and corrected. At least that is the theory. In practice of course the next stage usually fails because of undiscovered errors in the previous one, confusing matters and causing great aggravation. Nevertheless the implementation stages were as follows, albeit with a fair degree of backtracking:

(i) data structure design

(ii) run time system implementation and testing

(iii)Compiler extension and testing of new compiler

(iv) Testing of Graffiti programs compiled together with the run time system

(v) Separate compilation of run time system and further testing

(vi) Construction of device converters for additional devices

Stages (iv) and (v) differ in the fact that in (iv) the run time system is included as part of the user program. Compilation is thus longer, but error reports are easier to understand. In stage (v) the run time system has been compiled separately and the user program need only include declarations of the routines needed. The testing at stage (iv) tests whether the code generated by the compiler is correct, as the run time system should have been tested at stage (iii). Stages (v) and (vi) are really just refinements to the basic system, making it easier to use and able to produce pictures for various output devices. They are not necessary as far as the fundamental principles of Graffiti are concerned.

4.4. Data Structures

Before any coding can begin, it is necessary not only to have a good idea of the algorithms involved, but also of the representation of the data upon which the algorithms will be applied. It is important to do this thoroughly, otherwise the design of the code will be severely hampered and perhaps even made impossible. The need for a simple data

structure representing a picture is paramount. Given that the system is to work on small inexpensive computer systems, the structure has to be compact and still represent all the properties of pictures.

The picture is thought of as an ordered set of points, some connected to their successors, others not. Pictures themselves may be combined and connected with other pictures and they may be transformed. The structure naturally emerging is that of a binary tree on which a unique traversal is defined. Recursive pictures are not possible, as it would mean the traversal would never end. However there is no reason why the same subpicture should not occur several times, perhaps with some transformations performed upon it. It would be enormously wasteful of space if these were represented by separate structures. Instead the programmer is allowed to build a picture up of subpictures and rather than copying the subpicture every time a reference to the original is inserted into the tree. Thus the tree becomes a **Directed Acyclic Graph** (DAG for short)[38]. The structure is a graph because there may be several paths to the same substructure. It is directed because the ordering of the points uniquely defines a traversal and it is acyclic since recursive referencing is expressly forbidden, or rather, impossible. An example of a DAG is given in Fig. 4.1.

The pictorial information consists of point coordinates, transformations and combinations. It is also possible for a picture to contain text or a reference to a filed picture. Thus the following structures were defined:

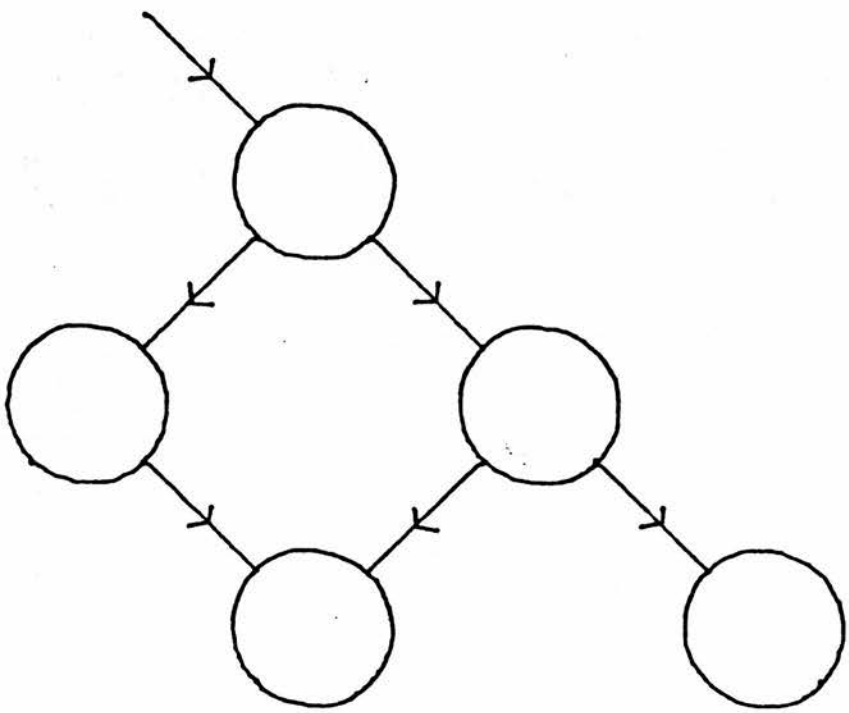


Fig 4.1: A Directed Acyclic Graph

```
structure       z.empty
let     empty = z.empty

structure z.point
        ( creal z.point.x, z.point.y, z.point.z )

structure z.trans
        ( cpntr z.trans.next; *c*creal z.trans.matrix )

structure z.include
        ( cpntr z.include.left, z.include.right )

structure z.connect
        ( cpntr z.connect.left, z.connect.right )

structure z.txt
        ( creal z.txt.x, z.txt.y, z.txt.z;
          cstring z.txt.string )

structure z.fil
        ( cstring z.fil.name )
```

The '*c*creal' is in fact a 4x4 matrix, such as identity below:

```
let identity = @1 of c*creal
               [
                @1 of creal [ 1.0, 0.0, 0.0, 0.0 ],
                @1 of creal [ 0.0, 1.0, 0.0, 0.0 ],
                @1 of creal [ 0.0, 0.0, 1.0, 0.0 ],
                @1 of creal [ 0.0, 0.0, 0.0, 1.0 ]
               ]
```

The result of a transformation is only defined if the given array is in fact a 4x4 array. It may be desirable to check that this is indeed the case, either at construction time or when the graph is traversed. If in the light of experience this proves to be so, it is easily done, however this detracts somewhat from the time efficiency of the algorithm.

These structures are constructed by the pictorial operators. The code generated by the compiler deals with obtaining storage and inserting the appropriate values in the right places. Only one structure is created with any one occurrence of the pictorial operators, but an expression may be as complex as the programmer is willing to type.

Examples

Ex. 1:

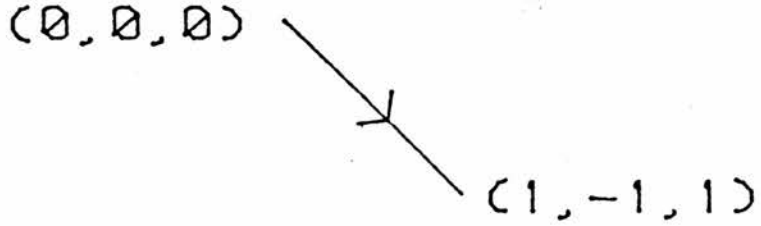
```
let pic1 = [0,0,0] ^ [1,1,1]
```

produces:

Ex. 2:

```
let pic2 = pic1 # rotate(0,0,pi/2)
```

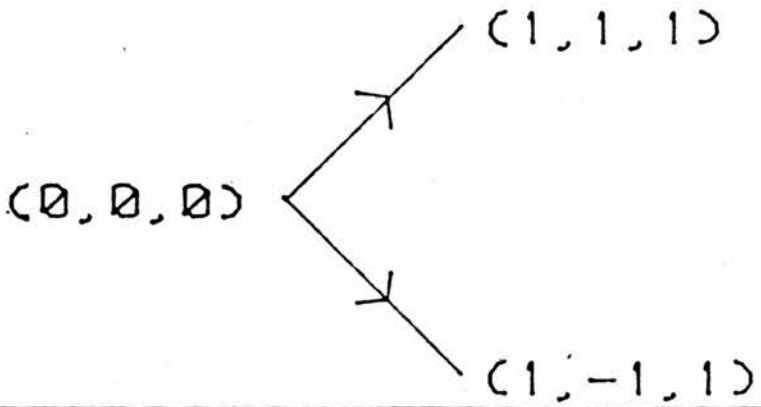
produces:



Ex. 3:

```
let pic3 = pic1 & pic2
```

produces:



5. The Graffiti Run Time System

5.1. Introduction

We deal here with the support routines for Graffiti, not the S-algol run time system, which is a subject in its own right and described elsewhere[39]. The Graffiti run time system consists of all those routines and functions necessary for the display of pictures once they have been created. The facilities provided have been listed before and in this section we deal exclusively with the questions of their implementation.

Given that we have a Directed Acyclic Graph to traverse and that the order of traversal is already defined, what remains? It is necessary to decide how the traversal procedure is to be implemented, which methods to use for perspective projection and clipping, how to organise the application of transformations to pictures and so on. All these questions are of vital importance, as the whole system may require so much storage space as to be totally useless on a small computer, or be so slow in execution that users might prefer pencil and paper. Although these problems do not invalidate the principles of Graffiti, they might result in the system being totally unusable. Obviously then the implementation decisions must be taken with care and with an eye for both space and time efficiency.

5.2. DAG Traversal

As the directed acyclic graph is essentially a tree with identical subtrees coalesced together, the methods of traversal applicable are those which can be used on trees. The construction of the picture and the ordering of points constrains the traversal to be in in-order, i.e. at any node the left subtree is traversed first, then the node itself is processed, then the right subtree becomes the target. Fig. 5.1 gives an example of in-order traversal if the nodes are visited in the numeric order specified. There can never be more than two subtrees, as the pictorial operators never take more than two operands.

There are only two major methods of traversing a binary tree in in-order, recursively and iteratively[38]. Both methods are similar in terms of speed, but the iterative method needs a constant amount of space, as there is no stack growth. The recursive method is trivial to program but requires stack space in proportion to the depth of the tree. I have decided to use the recursive method, as it is much easier to code, is easier to read and understand and I do not expect very deep trees. A properly balanced tree doubles in the number of points with each new level, so it seems unlikely that the stack growth will present a problem. There is one drawback, which is that the programmer should take care to balance the pictorial expressions if the number of points in a picture is to be large. Once again this decision is easy to change should it become necessary in the light of experience.

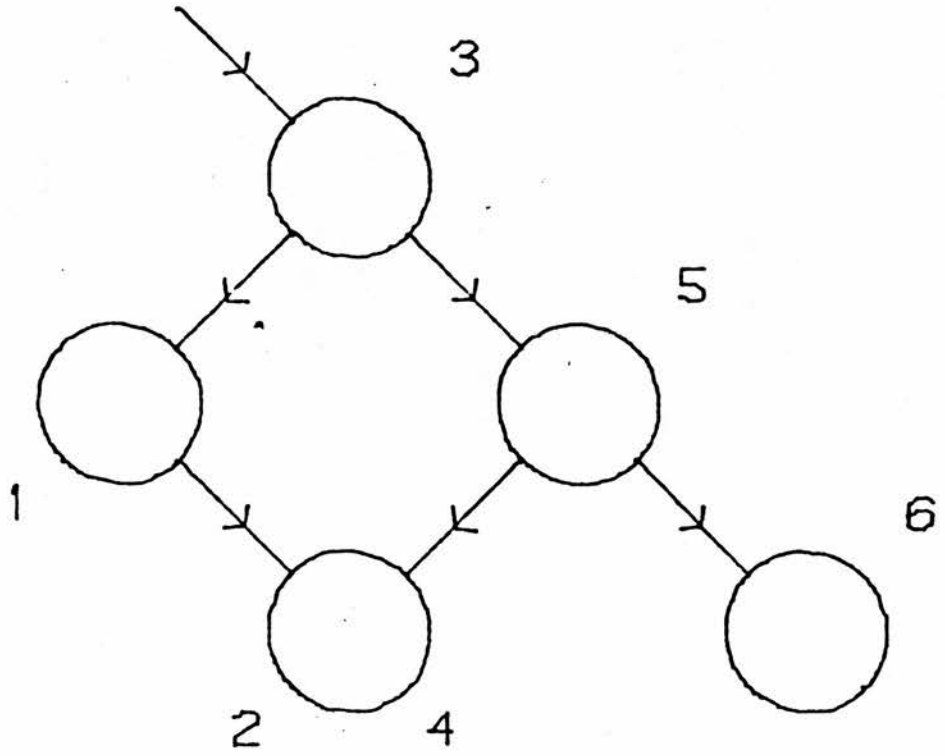


Fig 5.1: In-order Traversal

5.3. Windowing and Perspective

5.3.1. Windowing

The front face of the box, which is the one mapped onto the display, is treated as opaque except for a user defined window. By default the window is taken to be the whole front face of the box, but the user is able to change that. The user can also alter the size of the whole box and therefore inconsistencies can arise. There are three possibilities with arbitrary box and window limits:

1) The window is entirely contained by the front face of the box. This is the normal case and the picture will be clipped to the window limits.

2) The window and the front face of the box overlap only partially, i.e. there is a part of the window outside the front face. In this case the window is taken as the intersection of the window and the front face. A suitable error message is printed at plot time.

3) The window and the front face of the box do not overlap at all. Should this be the case, nothing is drawn and an error message is printed.

Therefore, although the screen coordinates are calculated using the box size, clipping is done as if the box were in fact determined by the window sizes in the x and y directions and the box z size.

5.3.2. Perspective

As perspective viewing is available, there are two different cases concerning clipping. If the user specifies that no perspective transform is to take place, the viewing box is just a cuboid. Should perspective transformation be necessary the cuboid becomes a frustrum of a pyramid. The back face of the box becomes effectively larger, since the perspective transform is basically division by the depth coordinate. Only after the transform has taken place does the box actually become a true cuboid. In actual fact it is the picture that is transformed, but the box is relative to the picture and the effect is the same - to bring more points at the back into the displayed image (see Fig. 5.2).

For the purpose of perspective transformation the user has to specify a distance from the centre of the front face of the viewing box at which the viewpoint is situated. It is not necessary to specify more, as any viewpoint can be brought onto a line perpendicular to the front face of the box by a suitable series of linear transformations. Thus we can assume the viewing is from the point V represented by:

[(xmin+(xmax-xmin)/2), (ymin+(ymax-ymin)/2), (zmin-dist)]

To do perspective and clipping easily it is conventional to transform the picture so as to have the viewpoint at [0, 0, 0]. Therefore a translation by -V is necessary. This translation is in fact achieved by the normal Graffiti transformation mechanism, i.e. a translation matrix is added to the picture being displayed.

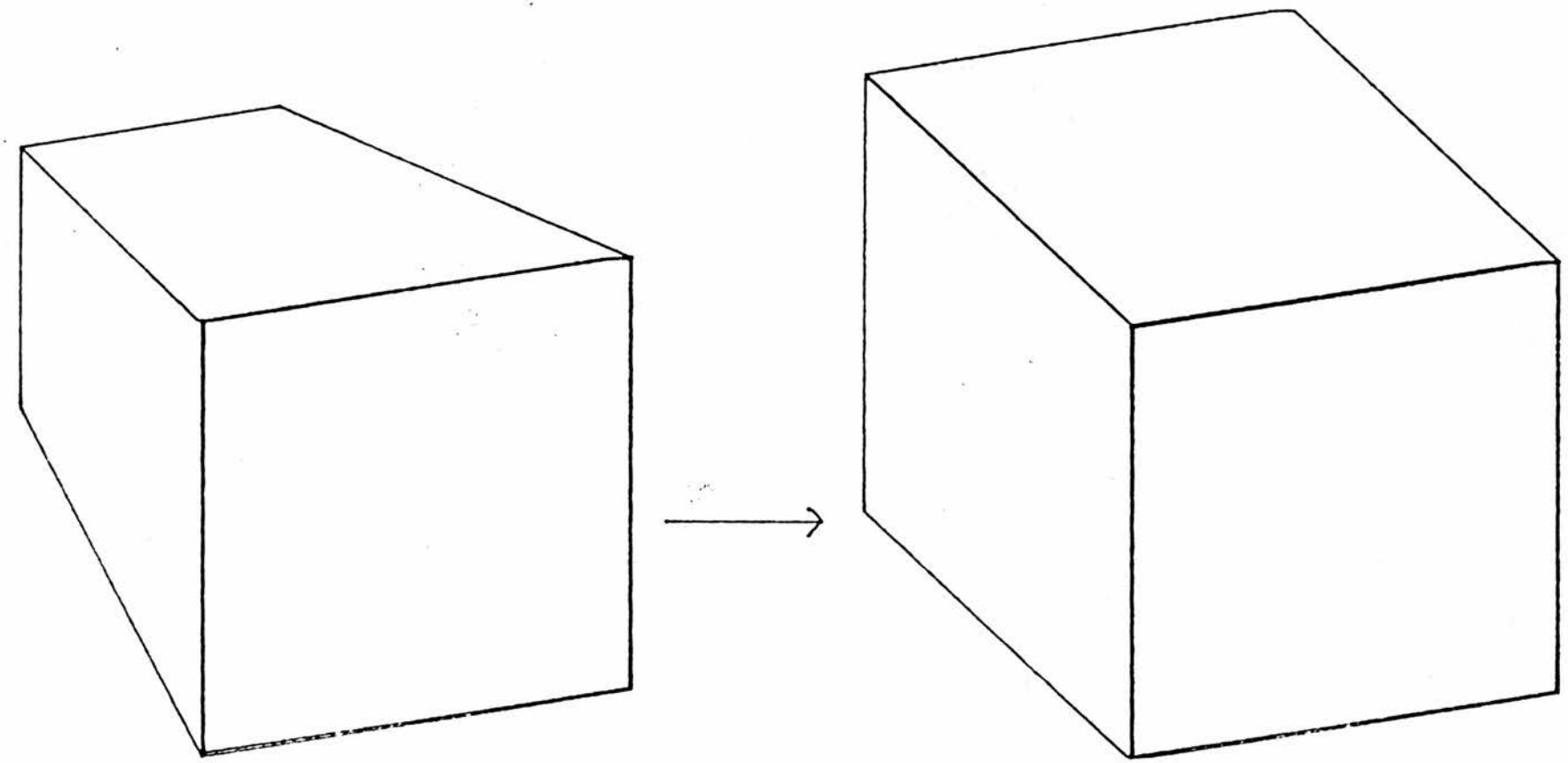


Fig. 5.2: Effect of Perspective Transformation

The important point about the above transformation, known as the **viewpoint transformation**, is that the z coordinates of all the points in the transformed picture are in fact the depth components of the distance of those points from the viewpoint. Therefore when the point coordinates emerge from the transformation routine, they can be clipped easily by the Sutherland and Cohen method[40].

The box and window limits must naturally also undergo the same transformation. We now have:

$$-xlim \leq x \leq +xlim$$

$$-ylim \leq y \leq +ylim$$

$$dist \leq z \leq +zlim$$

for the point to be visible, where dist is the distance specified by the user for the viewpoint distance, xlim and ylim are the transformed window limits and zlim is the transformed box depth.

5.3.3. Implementation

The clipping method mentioned is due to Sutherland and Cohen, but the details were obtained from Newman and Sproull[14]. The method has been implemented in hardware by Sproull and Sutherland[40] and a refinement is described by Sutherland and Hodgman[41].

It is best explained in its two dimensional form. The picture is divided into several regions, as in Fig. 5.3. These represent the various positions a point may have in relation to the viewport: above, below, left, right and indeed inside. The first four can partially

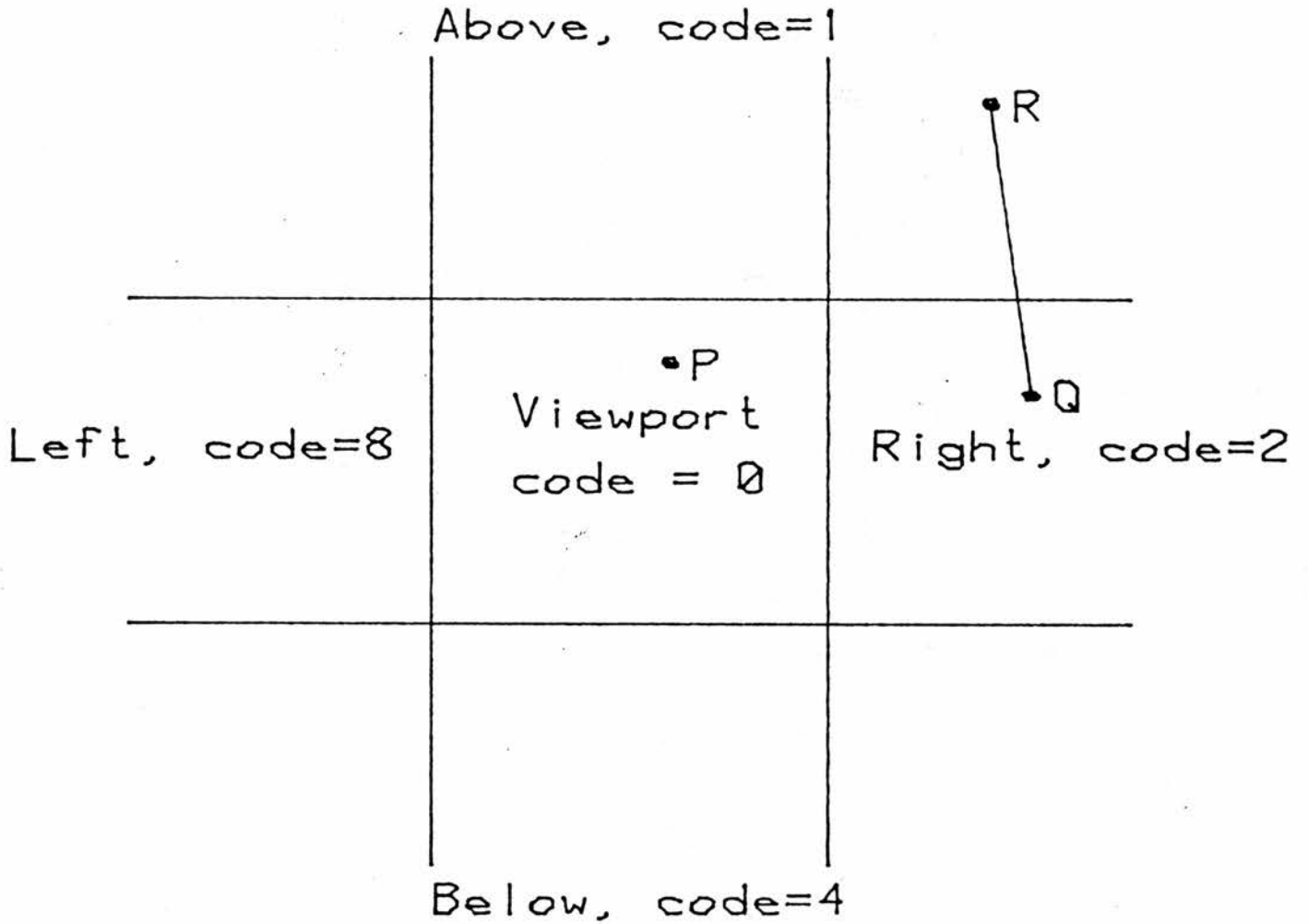


Fig. 5.3: Clipping Codes

overlap, for example a point may be both below and to the right of the viewport. Codes are normally allocated to the regions such that in a binary computer they would represent individual bits in a word, i.e. 1, 2, 4, 8. The area of the viewport is given the code 0. Then the code for any point is calculated by performing a logical **or** of all the area codes for this point. Thus point P in Fig. 5.3 would have the code 0, point Q would have the code 2 and point R would have the code 3. It now becomes trivial to decide whether a line is inside the viewport, as the codes of both of its end points must be 0. Similarly lines entirely to one side of the viewport are trivially rejected, because the logical **and** of the end points of any such line must be non-zero, e.g. line QR. The lines which can neither be trivially rejected nor accepted are one of two kinds. Either a portion of the line is visible or not. Both end points may be outside the viewport, or one may be inside. In both cases we seek to subdivide the line into more manageable portions. Sproull and Sutherland[40] in their hardware implementation use the binary chop technique. They divide the line into two equal portions and proceed to deal with each in turn according to the method described above. It is possible to implement this method fairly efficiently in hardware. A small number of subdivisions is enough to produce line portions which are either accepted or rejected. In the worst imaginable case, ten or so subdivisions would produce a line portion so small that it would be smaller than the screen resolution, at which point it can be ignored. However when the method is implemented in software the binary chop technique is surpassed in efficiency by a refinement which does not divide the line into two equal portions, but uses one of the area boundaries (above, below, etc) as the dividing point. Two unequal

portions are thus obtained, of which one must be either trivially rejectable or trivially acceptable. Moreover the maximum necessary number of divisions is four. The algorithm is easily adapted for use on a three dimensional frustrum by the addition of two further regions: **in front** and **behind.**

The efficiency arguments need not be true for all situations. In particular, if the computer system on which the implementation is to be attempted lacks floating point hardware, then it may well be more efficient to perform clipping by the binary chop. On the other hand, with the rapid development of hardware and the constantly falling costs we are now experiencing, it may already be possible to implement the second method in silicon as efficiently or better than the binary chop.

A second efficiency consideration arises at this point. When clipping and perspective transformations are taken together it is seen that there are two alternatives.

a) The perspective transformation is performed first upon every point, before clipping;

b) Perform the inverse of the perspective transform on the box limits, clip the points, perform the perspective transform on the remaining points and then check the z-coordinate limits and clip them too if necessary.

Method (b) is explained in Newman and Sproull[14] page 250 onwards. It involves multiplying every point by the aperture (the ratio of the distance from the eye to the viewport and the size of the viewport), clipping as before and then dividing all coordinates by the aperture.

Only then is the perspective transformation done on the remaining points. The relative efficiencies of the two methods depend on the number of lines completely outside the viewing box. If there are m lines altogether and n of them are not completely outside, then we have the following extra work done:

method (a)	method (b)
2m divisions	2n divisions
	+8n multiplications

All the other calculations have to be done in both cases. Thus we see that the choice on method depends on the sort of pictures the system will have to display, as well as the relative speeds of multiplication and division. In my view it is in any case desirable to produce simple pictures as quickly as possible, leaving the user to pay the penalty of displaying pictures with many more invisible lines than visible ones if he so wishes. Note that even with the divisions being four times as expensive as multiplications method (b) only becomes better if there are two or more times as many totally invisible lines in the picture as there are visible ones. Ideally of course, the system would be able to tell what sort of picture is being displayed and adapt accordingly.

The perspective transform itself is in fact very simple once the viewpoint transformation has been performed. It is not a linear transformation since a division by the depth coordinate is involved. Consider Fig. 5.4: The perspective division maps the point [x, y, z] to [x′, y′, z]. The depth (z) coordinate needs to be retained for the

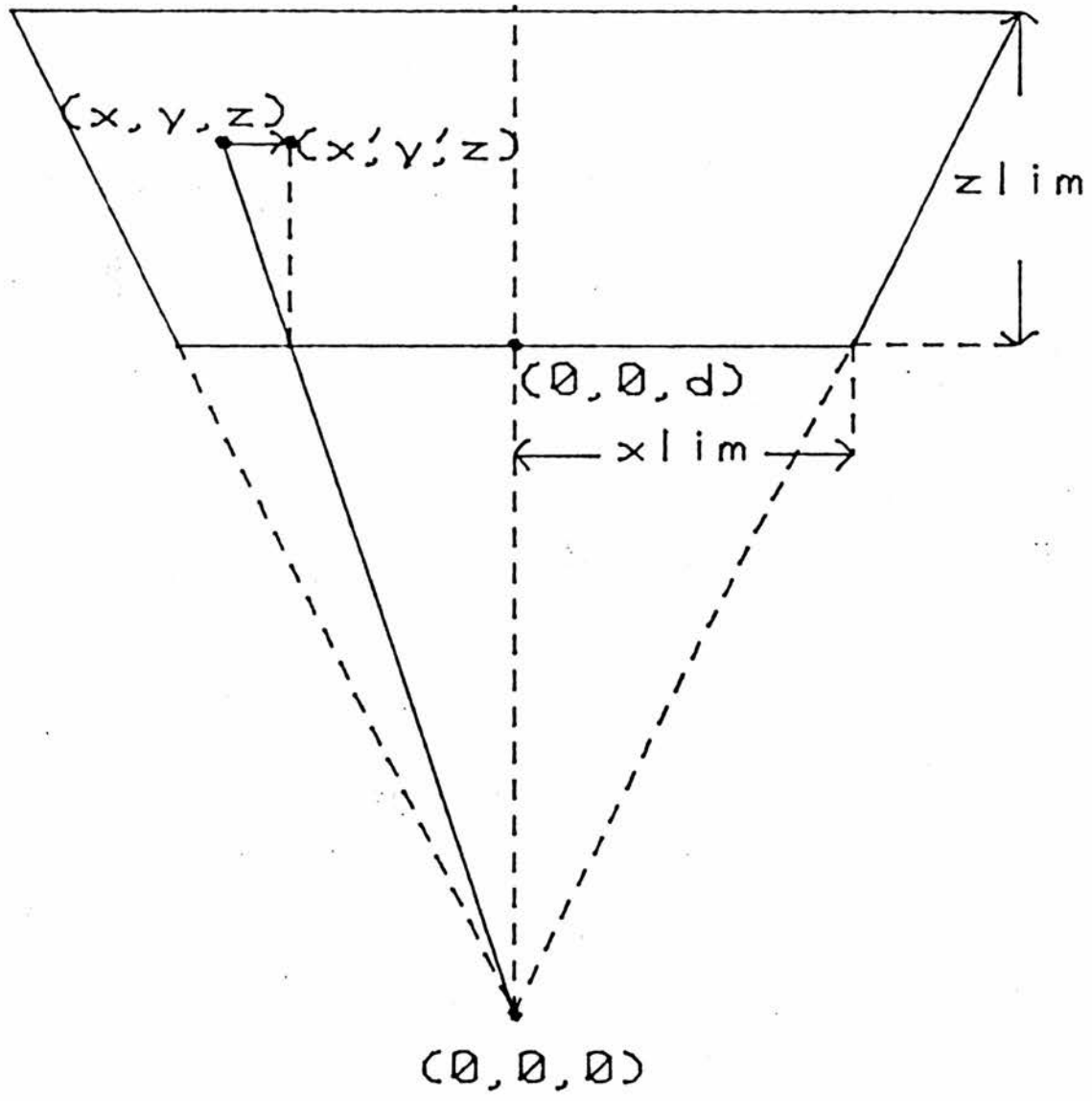


Fig. 5.4: Perspective Division

clipping algorithm.

We therefore have:

$$(x - x')/(z - d) = x / z$$

$$(x - x') = x [(z - d) / z]$$

$$x' = x - x [(z - d) / z]$$

$$x' = x[1 - (1 - d/z)]$$

Thus:

$$x' = x (d/z)$$

Similarly it can be shown that :

$$y' = y (d/z)$$

Perspective transformation and clipping together constitute the **viewing transformation** mentioned in chapter 3. The resulting low level picture description is two dimensional. The coordinate system is essentially the same as in the high level description, but with the origin at the viewpoint and with the z-coordinate always positive. This now has to be converted into a suitable device level description for display on some piece of hardware. For testing purposes the system produces Graphics Intermediate Format (GIF)[42]. A number of post-processors for GIF is available, including one for the Tektronix terminal. Approximations to the picture may also be printed out on an

ordinary line printer, thus reducing the debugging cycle of both the Graffiti internals and application programs. For efficiency reasons it will become necessary for Graffiti to produce output suitable for direct interpretation by a real device.

This completes the run time system implementation description. All that still remains to be described is the way in which the compiler was extended and integrated with the run time system.

6. Extending S-algol

6.1. Introduction

This chapter describes in detail how the S-algol language extensions were achieved. To be able to explain the changes to the compiler it is necessary first to give a rough outline of how the compiler works. The changes to the compiler are then explained. Finally it is shown how the whole system is fitted together, the compiler extensions enabling the users to generate the data structures representing their pictures and the run time system extensions allowing convenient display of the pictures.

6.2. The S-algol Compiler

The compiler is itself written in S-algol, It generates code for a non existent computer, the S-machine[39]. This computer would be very well suited for running S-algol programs, but unfortunately does not exist. The S-code therefore has to be interpreted. Interpreters currently exist for several real computers, with more in the pipeline. An S-algol interpreter in normally written in the assembly code of the target machine, thus achieving efficiency, but at some expense in terms of readability and portability. Nevertheless, being designed using stepwise refinement, the interpreter is relatively easy to rewrite for a different computer and operating system.

The compiler itself, being written in S-algol, is extremely portable. The only features that might change from system to system are routines associated with input and output. Any carefully written

program can easily be ported from one computer running S-algol to another. To port the compiler itself, the interpreter is rewritten and the machine dependent routines for the compiler adapted. The compiler is itself written using the top down techniques of stepwise refinement and carefully encapsulates any machine dependent parts in external modules.

S-algol programs are translated into S-code using the recursive descent technique[43]. This method, although very effective, has the property of blurring the distinction between the different phases of the compilation process. The phases are still there, even though they are all done at once. There are four conceptually separate phases through which a program must pass before ending up as S-code. They are:

lexical analysis
syntax analysis
type checking
code generation

(i) The **lexical analysis** phase recognises the various symbols and reserved words in the language and transforms them into tokens for subsequent processing by the syntax analyser.

(ii) The **syntax analysis** is concerned with the tokens passed to it by the lexical analyser. Constructs are checked for conformance with the grammar of the language. The S-algol grammar is context free.

(iii)The **type checker** must make sure that the language construction makes sense in terms of the types of expression used. Because the S-algol syntax is context free, it is possible to write down constructs which, although grammatically perfectly correct, nevertheless break the rules of type compatibility. The definition of the language consists of two parts, the context free syntax and the context sensitive type matching rules.

(iv) The **code generator** produces abstract S-code if the particular construction is accepted by all three of the preceding phases. As the S-machine is well adapted to S-algol, the code generator does not have a particularly difficult task.

The whole S-algol compiler, including comments, is about two thousand lines long.

6.3. Compiler Extensions

The extensions to the S-algol compiler made in order to turn it into a Graffiti compiler fall into two distinct categories. The first is in fact a single change, that of allowing a prelude file to be read in and compiled before compiling a user program. The second category consists of the changes made to enable the extended language features to be recognised.

(i) We deal with the prelude first. This change was done at St. Andrews by my supervisor for a Senior Honours project group. The effect of the prelude is to allow objects to be declared as if they were actually included at the beginning of a user program. The way

this works is by making the compiler read and start compiling from the prelude file. When the end of the prelude is reached the compiler, instead of finishing off, goes on to read and compile the user program. Any declarations appearing in the prelude thus behave just as if they were at the top of the program being compiled. This feature enables the predeclaration of the pictorial structures and is also necessary for the declaration of the standard routines without which a Graffiti program would not be able to run.

(ii) The language extensions turned out to be remarkably simple, once the structure of the compiler was understood. As all the mechanisms for analysis already existed in the compiler, all that was necessary was to invoke them in the right place at the right time when the Graffiti features were encountered. The line by line differences between the Graffiti compiler and the S-algol prelude compiler of October 1981 vintage are given in appendix C. The Graffiti changes consist exclusively of additions, as one would expect, since the Graffiti language is in fact a small superset of S-algol. They can be roughly classified as follows:

Declarations of new symbols for lexical analysis

Declarations of new types and type conversion operators

Specifications of actions to be taken when any of the new types and operators are encountered

All the additions are delimited in the compiler text with the comment lines:

```
! GRAFFITI BEGIN
and
! GRAFFITI END
```

The declarations of the new operator symbols and new reserved words appear towards the beginning of the compiler source. The reserved word symbols are used in the procedure **init1** which is part of the initialisations done by the compiler before it starts to read in the S-algol source text.

The type definitions **PIC** and **CPIC** are used in the procedure **pointer** as they are amongst the exclusive band of pointer type objects. They are also used in the procedure **type** as the types returned when the reserved words pic and cpic are encountered.

Procedure **standard.func** has the two conversion operators, pic.pntr.sy and pntr.pic.sy added.

Recognition of the point operator [x, y, z] is done in procedure **unary.op**, as the opening [behaves as a kind of unary operator operating on the list of coordinates.

Procedure **expr** was extended with a new subprocedure **pic.operation** which deals with all three operators, **&**, **^**and **#**. The priority of the connect and combine operations is 3, the same as for addition and subtraction, while that of transform is 4, just like multiplication. In the absence of brackets operations of equal priority are carried out from left to right, however this is completely transparent to the

programmer. **Expr** calls upon **pic.operation** if it encounters any of the three operators.

6.4. Integration

We have seen how the various components of the system perform their particular functions. It now remains to be shown how they are all fitted together into one integrated whole. The description below applies to the system state in February 1982.

The compiler takes the source program and translates it into S-code. The extensions to the compiler deal with the Graffiti operators and the pic data type. As the compiler generates code to create structures on the heap, these structures must be known to it. They are therefore declared in the prelude. All the other routines the user wishes to invoke must be included in his program at compile time. The same applies to all the global variables. This means that there is no need to call on the S-algol binder, unless the user uses other external routines. This also means that the user programs will take longer to compile. If the program is long this extra time will not be too difficult to ignore, but if the program is short the extra time delay is very frustrating. The next step in the development of the system is to move the declarations of all the global variables and constants into the prelude. After that the Graffiti run time support routines will be compiled externally. This should make Graffiti programs almost as quick to compile as ordinary S-algol programs. The difference will always be there because the prelude must also be compiled every time. The binder will then be a mandatory step for all Graffiti programs.

Therefore a typical Graffiti compile and run job on Unix might look like this:

```
G prog.G
Gbind LIB prog.out
g b.prog.out
```

G is the call of the Graffiti compiler, analogous to the S command. **Gbind** calls the S-algol binder with prog.out and the name of the Graffiti run time library as arguments. **g** calls up the Graffiti (i.e. S-algol) interpreter.

Various device drivers are provided. The first one that was implemented produced GIF[42] format for subsequent postprocessing. This has been useful for debugging purposes, because GIF is a human readable form thus enabling one to understand the output produced by Graffiti.

7. Conclusion

7.1. Introduction

This is the last of the chapters. It deals with the lessons learned during the project and looks ahead to possible new work on Graffiti.

7.2. The Current System

The system at present performs to specification. It used to be very cumbersome to use as it took a long time between calling up the compiler and getting out the picture. This is because the whole Graffiti run time code had to be compiled every time, as well as the fact that the resulting program produced GIF output which then had to be postprocessed. This has been very helpful in debugging the system, but would be unacceptable to users. Thus the system has been made easier to use, by precompiling the run time support code, and producing graphical device codes directly. In the first instance the Tektronix 4662[36] plotter was the target. Now device converter libraries exist for the Sigma GOC and the Anadex DP9501 printer as well as the Tektronix. Creating a new device converter library has proved to be a fairly easy task. The main difficulty lies in the decoding of the manufacturers' manuals and not in the writing of the code.

There do not appear to be any noticeable deficiencies in the implementation. Nevertheless the Graffiti system does require a fair amount of memory to be able to describe reasonable size pictures. The save and restore routines can be used to save pictures and call them up

later, but this is fairly inconvenient. On a computer system with a reasonable amount of memory, or even better with memory management giving a large virtual memory, the usefulness of the system would increase dramatically.

The system lacks features for graphical input and also for interrogating the pictorial data structures from the user program. This is not a deficiency in the implementation but rather it was deliberately left out of the design because it is not at all clear that a consistent general implementation is feasible.

Having started using Graffiti it was immediately apparent that it was grossly inefficient if the picture contained any transformations. This was because the algorithm for traversing the picture applied the transformation to the subpicture and then on return from deeper levels of recursion applied the inverse of the transformation. This resulted in twice as many matrix multiplications for every point, as well as several matrix inversions. The inversion was done away with using the recursive traversal itself to undo the transformation on the way back up the graph. This resulted in an improvement in speed by an order of magnitude. In particular a Hilbert curve which took over a minute to draw now takes less than fifteen seconds.

On further examination it was found that the matrix multiplications during the traversal, and the final multiplication of the point by the transformation matrix were much less efficient than they could be. This was mainly due to subscript evaluation. By first assigning a row of the matrix to a temporary variable a further speed improvement of five to ten percent was obtained.

It would appear that any improvement in execution speed beyond this cannot be achieved by coding in the run time system. Should it be necessary to increase the speed further two approaches can be pursued. One is to put the matrix into the language as a data type and code the interpreter to deal with it, the other is to obtain special purpose hardware to do matrix or vector operations. Both of these would totally destroy the portability argument while Graffiti is fast enough to keep up with the devices for which it is currently implemented.

7.3. Towards Better Graffiti

Extensions and additions to the system are already being planned. One display device that would be most convenient to display pictures on is the line printer. The resolution is atrocious, but pictures can be obtained quickly and cheaply for debugging purposes. The BBC microcomputer is a cheap way of providing colour graphics of medium resolution. An implementation of Graffiti using a BBC microcomputer as a graphics output device is currently under consideration.

Looking further ahead, two attractive possibilities present themselves. One is the implementation of hidden line removal. Techniques exist which are not particularly difficult to implement. The amount of processing required is still fairly large and this has been the main reason for not attempting it as part of the system so far. Displaying a picture with hidden lines removed could easily be incorporated into Graffiti as just another of the modes of display.

The second and very attractive possibility is the implementation of the system on a dedicated computer. One that has been in my mind for a

while is the LSI11 from DEC. The Department of Computing Science at Glasgow has had one for a while, but the relatively small memory size has kept me from doing anything about it. The advent of the ICL PERQ and its advanced graphics capabilities has shifted my attention to it. It would be most interesting to transfer Graffiti to the PERQ. It is basically a single user machine with a high resolution graphics display and an operating system based on UNIX.

7.4. Epilogue

Well here it is. Would I have done it differently if I had to start again? The answer is a firm 'probably'. Certainly holding down a full time job while completing a project is not a good idea and I would have liked to avoid it. As for the Graffiti system I still think there was a need for a simple to use system with basic facilities, based on sound principles and going hand in hand with a powerful programming language. I feel that this project has gone at least some of the way to providing such a system and leave it to others to say just how successful it has been.

References

[1]. WIRTH, N. AND HOARE, C.

"A Contribution to the Development of Algol", CACM, Vol. **9**, (1966).

[2]. SMITH, D. N.

"GPL/1 - a PL/1 extension for computer graphics", pp. 511-528 in Proc. AFIPS (SJCC), (1971).

[3]. MORRISON, R.

"Algol R", Report No. CS/78/1, St. Andrews University, Department of Computational Science (1978).

[4]. MORRISON, R.

"S-algol Reference Manual", Report No. CS/79/1, St. Andrews University, Department of Computational Science (1979).

[5]. RITCHIE, D. M. AND THOMPSON, K.

"The UNIX Timesharing System", CACM, Vol. **17**, (7)(1974).

[6]. RITCHIE, D. M. AND THOMPSON, K.

"The UNIX Time-Sharing System", Bell Sys. Tech. J., Vol. **57**, (6) pp. 1905-1929 (1978).

[7]. FREEMAN, H.

"Computer Processing of Line Drawing Images", ACM Computing Surveys, Vol. **6**, (1)(1974).

[8]. MORRISON, R. AND PODOLSKI, Z.

"The Graffiti Graphics System", in proc DECUS UK Conference 1978, , Bath, England (1978).

[9]. PODOLSKI, Z., GREENWELL, H., AND MCGILL, A.

"The Graffiti Plotting System", Senior Honours Project Report, St. Andrews University, Department of Computational Science (1977).

[10]. "General Methodology and proposed Standard for a CORE Graphics System", Computer Graphics (SIGGRAPH-ACM), Vol. **11**, (3)(1977).

[11]. "State-of-the-art of Graphic Software Packages", Computer Graphics (SIGGRAPH-ACM), Vol. **11**, (3)(1977).

[12]. WILLIAMS, R.

"A Survey of Data Structures for Computer Graphics", ACM Computing Surveys, Vol. 3, (1)(1971).

[13]. SUTHERLAND, I. E.

"Sketchpad: a man-machine graphical communication system", in Proc. SJCC, Spartan Books, Baltimore, Maryland (1963).

[14]. NEWMAN, W. AND SPROULL, R.

Principles of Interactive Computer Graphics (2nd ed), McGraw - Hill (1979).

[15]. ROGERS, D. F. AND ADAMS, J. A.

Mathematical Elements for Computer Graphics, McGraw - Hill (1976).

[16]. CHILDS, D. L.

"Description of a Set Theoretic Data Structure", in Proc. AFIPS (FJCC), (1968).

[17]. BRACCHI, G. AND SOMALVICO, M.

"An Interactive Software System for CAD, an Application to Circuit Design", CACM, Vol. **13**, (9)(1970).

[18]. BRACCHI, G. AND FERRARI, D.

"A Language for Treating Geometric Patterns in a 2D Space", CACM, Vol. **14**, (1)(1971).

[19]. JONES, W. R.

"A Macro Facility for Interactive Display", Software Practice and Experience, Vol. **1**, (1971).

[20]. GREENBERG, D. P.

"An Interdisciplinary Laboratory For Graphics Research and Applications", in Proc. 4th annual conference on Computer Graphics, Interactive Techniques and Image Processing, (1977).

[21]. MILLER, W. F. AND SHAW, A. C.

"Linguistic Methods in Picture Processing - a Survey", in Proc. AFIPS (FJCC), (1968).

[22]. STANTON, R. B.

"The Interpretation of Graphics and Graphics Languages", in Graphic Languages, ed. Rosenfeld & Nahe, North Holland (1972).

[23]. WARNER, J. R.

"MIDAS - A Compositional Modelling System", in Proc. AFIPS (SJCC), (1977).

[24]. RICCI, A.

"A Constructive Geometry for Computer Graphics", Computer Journal, Vol. **16**, (2)(1973).

[25]. DIN, Graphical Kernel System (GKS), proposal of standard DIN 00 66 252, West German Standards Institute (1978).

[26]. ROSENTHAL, D. S. H.

Models of Interactive Graphics Software Working document 30 of the British Standards Institute DPS13 working group 5 on graphics standards 1979.

[27]. NEWMAN, W. M. AND VAN DAM, A.

"Recent Efforts Towards Graphics Standardisation", Computing Surveys, Vol. **10**, (4)(1978).

[28]. KILGOUR, A. C.

"A Hierarchical Model of a Graphics System", Computer Graphics (SIGGRAPH-ACM), Vol. **15**, (1)(1981).

[29]. SINT, MARLEEN

"Design of an Algol-68 Extension for Computer Graphics", Computer Graphics (SIGGRAPH-ACM), Vol. **14**, (4)(1980).

[30]. HAGEN, T., TEN HAGEN, P.J.W., KLINT, P., AND NOOT, H.
"The Intermediate Language for Pictures", in Information Processing 77, ed. B. Gilchrist, North Holland (1977).

[31]. KEITH, S. R.
"A transformation Structure for Animated 3D Computer Graphics", Computer Graphics (SIGGRAPH-ACM), Vol. 15, (1)(1981).

[32]. BERK, T. AND KAUFMAN, A.
"Roster of Graphic languages and General Subroutine packages", Computer Graphics (SIGGRAPH-ACM), Vol. 14, (4)(1980).

[33]. COLE, A. J. AND MORRISON, R.
An Introduction to Programming With S-algol, Cambridge University Press (1982).

[34]. NOTLEY, M. G.
"A Graphical Picture Drawing Language", Computer Bulletin, Vol. 14, (3)(1970).

[35]. LARKIN, F. M.
"A Graphical Output Language and its Implementation", Report CLM-P 139, UKAEA Culham ().

[36]. 4006-1 Computer Display Terminal Users Instruction Manual, Tektronix, Inc., Beaverton, Oregon, USA (Rev A, 1976).

[37]. KERNIGHAN, B. W. AND PLAUGER, P. J.
Software Tools, Addison-Wesley (1976).

[38]. HOROWITZ, E. AND SAHNI, S.

Fundamentals of Data Structures, Computer Science Press (1977).

[39]. BAILEY, P. J., MARITZ, P., AND MORRISON, R.

"S-algol Abstract Machine", CS/80/2, University of St. Andrews, Computational Science Department (1980).

[40]. SPROULL, R. F. AND SUTHERLAND, I. E.

"A Clipping Divider", in AFIPS (FJCC), Thompson Books, Washington D.C. (1968).

[41]. SUTHERLAND, I. E. AND HODGMAN, G.

"Reentrant Polygon Clipping", CACM, Vol. **17**, p. 32 (1974).

[42]. PROSSER, C.

Graphical Output Methods And Their Relation To Display System Design, M.Sc. Thesis, University of Glasgow, Computing Science Department (1981).

[43]. DAVIE, A. J. T. AND MORRISON, R.

Recursive Descent Compiling, Ellis Horwood, Chichester (1981).

Appendix A: Syntax and Type Matching Rules

These rules specify the extensions to the S-algol syntax and type matching. They are in addition to the normal rules as set down in the S-algol Reference Manual.

A.1. Syntax Extensions

```
<exp5> ::= <pic.expression> | (as before)
<pic.expression> ::= <pic.exp1> [ <transop> <exp> ]*
<pic.exp1>       ::= <pic.exp2> [ <combop> <pic.exp2> ]*
<pic.exp2>       ::= <bra> <exp> , <exp> , <exp> <ket>
<exp>            ::= <expression>
<bra>            ::= [
<ket>            ::= ]
<transop>        ::= #
<combop>         ::= <connect.op> | <include.op>
<connect.op>     ::= ^
<include.op>     ::= &
<standard.id>    ::= pic.pntr | pntr.pic | (as before)
<type>           ::= pic | (as before)
```

A.2. Type Matching Extensions

Again, these rules are in addition to the ones in the S-algol reference manual.

```
{ pic } < combop > { pic }                    =>    { pic }
{ pic } < transop > { *c*creal }               =>    { pic }
```

Appendix A

< bra > { real },{ real },{ real } < ket > => { pic }

pic.pntr ({ pic }) => { pntr }

pntr.pic ({ pntr }) => { pic }

Appendix B: Pictorial Data Structures

These are the structures built up by the operators:

```
structure       z.empty
let     empty = z.empty

structure z.point
        ( creal z.point.x, z.point.y, z.point.z )

structure z.trans
        ( cpntr z.trans.next; *c*creal z.trans.matrix )

structure z.include
        ( cpntr z.include.left, z.include.right )

structure z.connect
        ( cpntr z.connect.left, z.connect.right )

structure z.txt
        ( creal z.txt.x, z.txt.y, z.txt.z;
          cstring z.txt.string )

structure z.fil
        ( cstring z.fil.name )
```

Appendix C: Compiler Changes

The compiler changes are in two parts, those in the main compiler program and those in the external definitions file which contains the machine specific routines. The differences are here given in UNIX diff format. In order for the listing to fit on the A4 page width some lines have had to be broken.

Lines of the form '23a40,50' indicate that lines 40 to 50 of the new program are in addition to line 23 of the old program. Usually some lines prefixed with a '>' follow, which means they are to be inserted in the old program text.

A line of the form 23,30c40,50 indicates that lines 23 to 30 of the old program are to be replaced by lines 40 to 50 of the new. Lines to be removed from the old program are prefixed by a '<' in the listing.

Where lines have had to be broken to fit the page, they start with an indentation.

C.1. Main compiler program changes

```
3a4
> %lines=62
59a61,65
> ! GRAFFITI BEGIN
> let include.sy = "&" ; let connect.sy = "^" ;
        let transform.sy = "#"
> ! GRAFFITI END
```

```
>
>
82a89,95
> ! GRAFFITI BEGIN
> let pic.sy = "pic" ; let cpic.sy = "cpic"
> let text.sy = "text"
> let pic.pntr.sy = "pic.pntr" ; let pntr.pic.sy = "pntr.pic"
> ! GRAFFITI END
>
>
105a119,123
>
> ! GRAFFITI BEGIN
> let PIC = scalar( pic.sy ) ; let CPIC = const( PIC )
> ! GRAFFITI END
>
138c156,160
< @1 of cstring[ case.sy,cstring.sy,cint.sy,creal.sy,
        cfile.sy,cpntr.sy,cbool.sy,"" ],
---
> @1 of cstring[ case.sy,cstring.sy,cint.sy,creal.sy,
        cfile.sy,cpntr.sy,cbool.sy,
> ! GRAFFITI BEGIN
>                                                cpic.sy,
> ! GRAFFITI END
>                                                    "" ],
151c173,177
```

Appendix C

```
< @1 of cstring[ procedure.sy,pntr.sy,peek.sy,"" ],
---
> @1 of cstring[ procedure.sy,pntr.sy,peek.sy,
> ! GRAFFITI BEGIN
>           pic.sy, pic.pntr.sy, pntr.pic.sy,
> ! GRAFFITI END
>                          "" ],
155c181,185
< @1 of cstring[ to.sy,then.sy,"" ],
---
> @1 of cstring[ to.sy,then.sy,
> ! GRAFFITI BEGIN
>                text.sy,
> ! GRAFFITI END
>                "" ],
257a288,290
> ! GRAFFITI BEGIN
> PIC, CPIC,
> ! GRAFFITI END
644,645c677,683
< let heading := "S-algol System" ; let eot = "?"
< let lines.per.page := 44 ; let lines.on.page := lines.per.page + 1
---
> ! GRAFFITI BEGIN
> ! was : let heading := "S-algol System" ; let eot = "?"
> let heading := "Graffiti Graphics System" ; let eot = "?"
> ! GRAFFITI END
```

```
>
> let lines.per.page := 62
        ! always 4 less than the page length (ZP 9.1.82)
> let lines.on.page := lines.per.page + 1
653c691
<           else write "'n**** Program Compiles ****'n"
---
>           else if listing do
            write "'n**** Program Compiles ****'n" ! ZP 9.1.82
1039c1077
<                       lines.per.page := the.lit }
---
>                       lines.per.page := the.lit-4 } ! ZP 9.1.82
1192a1231,1234
> ! GRAFFITI BEGIN
> pic.sy        : { next.sy ; PIC }
> cpic.sy       : { next.sy ; CPIC }
> ! GRAFFITI END
1544a1587,1590
> ! GRAFFITI BEGIN
>       pic.pntr.sy : { arg1( PIC ) ; PNTR }
>       pntr.pic.sy : { arg1( PNTR ) ; PIC }
> ! GRAFFITI END
1638a1685,1702
> ! GRAFFITI BEGIN
>  lsb.sy : begin
>            next.sy
```

```
>           let level = ssp ; let plevel = psp
>           let t = lookup( "z.point", true )
>           match( REAL, clause )
>           reverse.stack( REAL )
>           mustbe( comma.sy )
>           match( REAL, clause )
>           reverse.stack( REAL )
>           mustbe( comma.sy )
>           match( REAL, clause )
>           reverse.stack( REAL )
>           mustbe( rsb.sy )
>           form.structure( level, plevel )
>           PIC
>            end
> ! GRAFFITI END
1727a1792,1805
> ! GRAFFITI BEGIN
>      procedure pic.operation( cstring op.type; cint priority )
>      if le2( n, priority ) do
>      begin
>           let level = ssp ; let plevel = psp-st.size
>           match( PIC, t )
>           let temp = lookup( op.type, true )
>           match( (if op.type = "z.transform"
                                then VECTOR(VECTOR(REAL))
>                               else PIC), expr(priority+1))
>           form.structure( level, plevel )
```

```
>           t := PIC
>       end
> ! GRAFFITI END
>
1768a1847,1853
>
> ! GRAFFITI BEGIN
>       include.sy      : pic.operation( "z.include", 3 )
>       connect.sy      : pic.operation( "z.connect", 3 )
>       transform.sy      : pic.operation( "z.transform",4)
> ! GRAFFITI END
>
2137c2234
< if error do { let so = Create.code ; close( so ) }
---
> if error do { let so = Create.code ;
        close( so ) ; abort } ! ZP 9.1.82
```

C.2. External Procedure Changes

This boils down to opening the prelude file containing the global declarations of the Graffiti structures, global variables, etc.

```
111a112,119
>
> procedure Open.prelude( -> file )
```

```
> begin
>     let s = "/usr/lib/G/pic.defs"
>     let s1 = open( s,0 )
>     if s1 = nullfile do
      { write "'nCannot open file of standard definitions called",
           s,"'n" ; abort }
>     s1
> end
```

Appendix D: GRAFFITI User Manual

D.1. Introduction

The Graffiti graphics system is designed to offer users convenient three dimensional graphics facilities. It is a simple general purpose graphics system, in which may be embedded a more sophisticated one. The facilities of Graffiti form a set of extensions to the S-algol programming language (1). S-algol is a general purpose programming language in the Algol family, with data and control structures which enable the user to build up the required level of abstraction in his program.

Graffiti provides for creation, manipulation and output of pictures. These objects form a new data type, called pic, which have full "civil rights" in S-algol. One can have pic expressions, arrays, variables, constants and procedures. Objects of type pic can be passed as procedure parameters, and they can be the fields of structures. A pic is a representation of a picture kept in store, allowing convenient scaling, translation, rotation and other transformations. A picture is described in terms of points, other pictures and transformations on pictures.

The pictures are defined in a three-dimensional logical space. The users will specify which part they wish to display, by defining a "box". Only those parts which are within the box will be drawn when it is directed to a display device. Lines protruding outside will be clipped appropriately. It is not an

error condition for clipping to occur, however a message may be printed the first few times it happens.

The coordinate system used is the left handed cartesian one. The x values increase towards the right, the y values upwards and the z values increase away from the eye. The further away a point is, the larger its z coordinate.

To be able to use Graffiti effectively, the user will have to become familiar with the S-algol programming language, as described in (1).

D.2. Pictures

As with any other data type, a pic variable must be declared and initialised before it can be used. The declaration takes the form:

let name := < picture expression >

The pic called "name" is declared, and its initial value is the expression.

The value of a pic variable may be altered by an assignment statement. The form of the assignment statement is :

< picture variable > := < picture expression >

Named constants of type pic may also be declared.

There are several kinds of < picture expression > . An example is the identifier "name". Expressions may further consist of a procedure of type pic, a point or an expression formed from any of the above using pictorial operators.

Pictures of points may be created from three real expressions as follows:

[x, y, z]

There are standard procedures of type pic which have pictures as their values. In addition to those, the user may declare his own pic procedures.

D.3. Pictorial Operators

Pictures may be combined using the following two operators :

& the include operator;

^ the connect operator.

Include takes two pictures as its operands, and creates a new one containing them both. An example is:

```
pic1 & pic2
```

Connect is similar to include , but it also adds a line from the last point of the left picture to the first point of the right picture. Thus lines may be created from points, and more complicated pictures from less complex ones.

eg.

```
pic1 ^ pic2
```

An important fact to remember is that points within a picture are ordered. This order depends on the manner the picture was described. When using the two operators, the first point of the left operand always becomes the first point, and the last point of the right operand the last point, of the resultant picture. There is no reason why the two points may not share the same display position. An example of that is a picture of a closed loop.

An example :

```
let pic1 := [ 0.0 , 0.0 , 0.0 ]
    pic1 := pic1 ^ [ 1 , 0 , 0 ] ^ [ 1 , 1 , 0 ] ^
            [ 0 , 1 , 0 ] ^ pic1
```

The picture "pic1" is declared, and in the next statement is given the value of a point at the origin. Using the connect operator, "^", pic1 is connected to points [1,0,0], [1,1,0], [0,1,0] and then back to the origin. This will display as a square. (See fig. 3.1). Note that the first and last points of this picture coincide at the origin. Whether pic1 is to be connected to another picture, or another picture to it, the point of connection will be the same.

eg.

```
let pic2 := pic1 ^ [ 2, 1, 0 ]
```

results in a square plus a line from [0,0,0] to [2,1,0], (fig. 3.2) whereas

```
let pic3 := [ 2, 1, 0 ] ^ pic1
```

creates a line from [2,1,0] connected to the square at the point [0,0,0](fig. 3.3). In the first case the resultant picture has as its last point [2,1,0], while in the second case this is the first point.

The two operators have the same precedence, and will be evaluated from left to right in an expression containing them both. Brackets may be used in the normal manner to change the order of evaluation.

There is another pictorial operator. It is used to transform pictures. The operator takes a picture and a 4x4 matrix and returns a picture. It is

the transform operator.

The matrices used to represent transformations must be invertible, i.e. they must have a non-zero determinant.

eg.

```
pic1 # matrix
```

In a more complicated expression involving several transform operators, the transformations are done left to right. Parentheses may be used to change this.

To facilitate the use of this very powerful, but also very general mechanism, several standard functions of type *c*creal are provided. They all return 4x4 matrices as their values. The more common transformations are amongst them. Thus for all but the most complex uses, no knowledge of matrices is necessary. However, the full generality is available should anyone wish to use it. The matrices are normal two dimensional real arrays in S-algol, but they must contain exactly four rows of four entries each for use with the transform operator.

The precedence of the transform operator is higher than that of ^ and &. Thus

```
[ x, y, z ] ^ [ a, b, c ] # matrix
```

is equivalent to :

```
[ x, y, z ] ^ ( [ a, b, c ] # matrix )
```

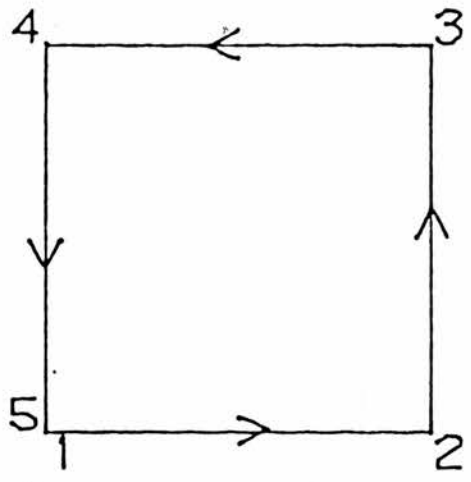


Figure 3.1

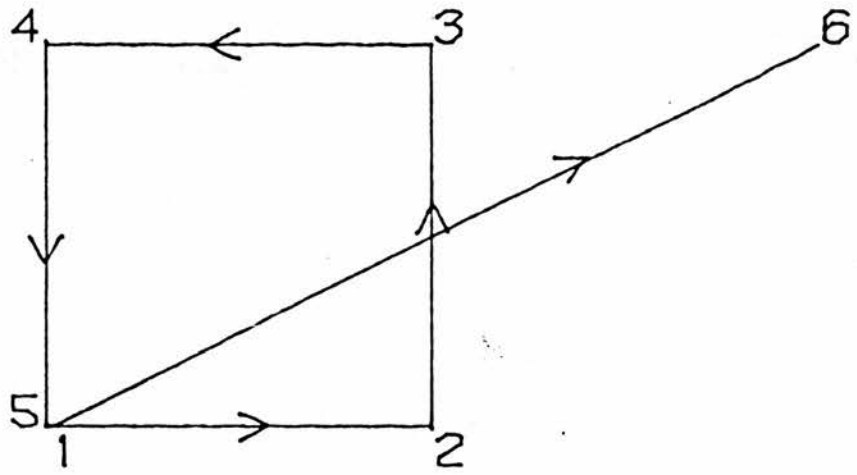


Figure 3.2

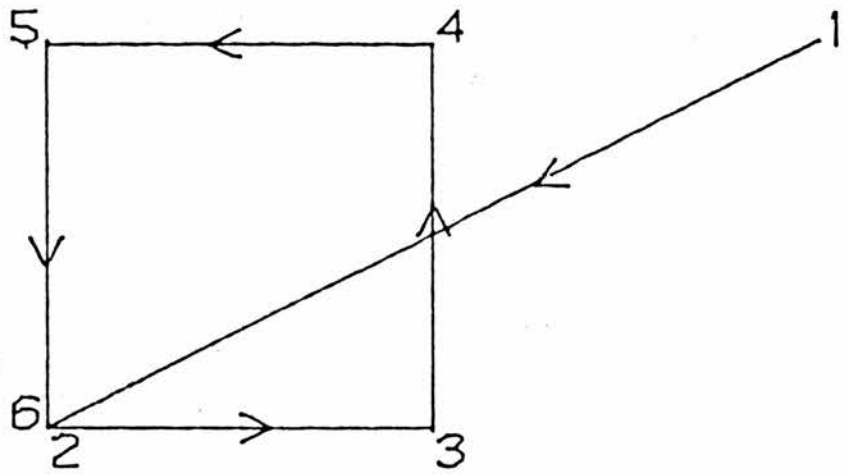


Figure 3.3

D.4. Standard Functions

Apart from the S-algol standard functions, Graffiti will have some of its own. Three of them are concerned with transformations. They are of the type procedure (real, real, real -> *c*creal). Their values are 4x4 matrices, which can be manipulated in the normal manner within S-algol, but their main use is expected to be with the # operator. The functions take three real values as their parameters, representing the relevant transformations with respect to the x, y and z axes. The functions are :

procedure move (real x, y, z -> *c*creal)

returns a 4x4 matrix representing a translation by the required amounts in the x, y and z directions;

procedure scale (real x, y, z -> *c*creal)

returns a matrix representing scaling by x in the x coordinate, and y and z in the y and z coordinates respectively. The effect of scaling is to multiply every coordinate value in a picture by those amounts;

procedure rotate (real x, y, z -> *c*creal)

returns a matrix representing a rotation by x about the x-axis, y about the y-axis and z about the z-axis. The rotations are in radians;

The transformations can each be thought of as composed of three sub-transformations, which are done from left to right. This is only important in

the case of rotation, because rotations are not generally commutative, and the order in which they are performed matters.

Rotation is performed in the clockwise direction when looking along the axis in the direction of increasing coordinates. When objects are rotated about the z-axis, the points appear to move clockwise. Rotation about the x-axis has the effect of bringing those points with positive y coordinates nearer and those with negative y coordinates are moved further away. A similar effect is observed when rotating about the y-axis, but here it is the sign of the x coordinates that determines whether points are brought nearer or moved further away.

Scaling a picture has the effect of multiplying every coordinate value in it by the parameters. When scaling it should be borne in mind that scale factors greater than 1 increase the size of the picture, and those between 0 and 1 decrease it. Scaling by 0 is not allowed, as it results in an irreversible change. When negative numbers are used, the effect is the same as with positive ones, but the picture is also reflected in the respective axis.

An example :

```
let p1 := [0,0,0]
p1 := p1 ^ [1,0,0] ^ [1,1,0] ^ [0,1,0] ^ p1
let p2 := p1 # scale( sqrt(2), sqrt(2), 1 )
p2 := p2 # move( -1/sqrt(2), -1/sqrt(2), 0 )
p2 := p2 # rotate( 0, 0, pi/4 )
p2 := p2 # move( 0.5, 0.5, 0 )
```

```
plot( p1 & p2 )
```

This program draws a picture of a square of unit side, containing a square of side 1/sqrt(2) touching the centre points of the sides of the first one. The squares are centred on the point [0.5,0.5,0], as in figure 4.1.

There is a standard function which makes a picture out of some text. It is intended to use the existing hardware character generators in any given output device. Later it might be feasible to use a number of different predefined character sets, or even to allow the users to specify their own. This function is called "text" and it requires a string and three numbers specifying the starting point of the text. The text will always be horizontal, but the starting point will be affected by the three transformation routines. Any control characters in the displayed text will have hardware dependent effect.

```
procedure text ( real x, y, z; string lines -> pic )

    creates a picture which consists of the lines of text ;
```

There will also be a standard function called "restore". It is described in detail in a further chapter, together with the related routine "save". Save and restore are used for output and input of pictures in unevaluated form.

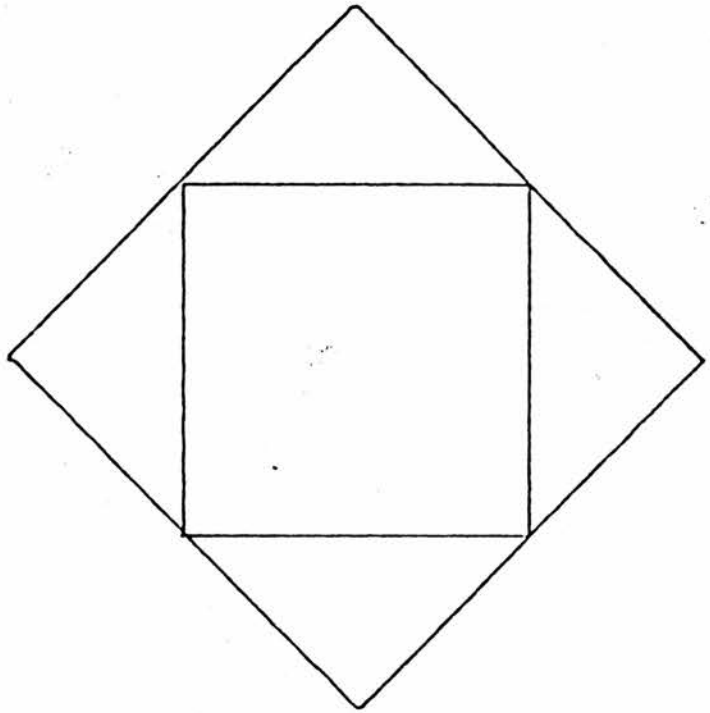


Figure 4.1

D.5. Modes of Display

So far we have considered the creation of pictures. To be able to use a graphics system it is also necessary to have the ability to direct these pictures to a display device. There are many different display types available, and the way a picture is displayed will be dependent on the particular device used. For example, some devices will allow the drawing of coloured lines, others not. Graffiti attempts to keep these device-dependent features separate from the actual pictures. Thus pictures are treated as just ordered sets of points, some of which may be connected to their successors, others not. Any other possible "attribute" that pictures might have is actually a function of the way they are displayed. These attributes are called display modes in Graffiti.

These are the routines to select various display modes:

procedure set.new.frame (bool new.frame)

default value is to select a new frame every time a picture is displayed.
new.frame = true implies selecting new frame,
new.frame = false implies superimposition on old frame;

procedure set.frame.border (bool border)

border = true implies that a border is to be drawn with any displayed picture,
border = false stops this.
The default value is false. The border is drawn according to the

limits of the front face of the "box";

procedure set.window.border (bool border)

similar to above, but draws border around the window area within the frame. If the window overlaps only partially with the frame, it is truncated to the frame border. If there is no overlap nothing is drawn;

procedure set.limits (real xmin, xmax, ymin, ymax, zmin, zmax)

limits of the box outside which nothing is visible. The front face maps as large as possible onto the display. The displayed picture is centred on the display.
Default values are (0,1, 0,1, 0,1) ;

procedure set.window.limits (real xminw, xmaxw, yminw, ymaxw)

default values are (0,1, 0,1).
This procedure opens a window on the front face of the box, through which viewing takes place. Thus it is easy to draw on only a small part of the display, leaving the rest for interaction, or for other pictures.
If the values specified are inconsistent with the set.limits values nothing will be drawn, and an error message produced. However partial overlap is possible;

procedure set.perspective (real distance)

this sets the distance from which the box is to be viewed. It is

measured at right angles to the centre of the front face of the box, i.e. the face with the smaller z coordinate.

If the value given is zero this indicates no perspective effects. This is the default. The transformation will give the best approximation to true perspective if the display is viewed from the same distance (in the same units) as was specified in the command;

procedure set.line.style (int style)

Default style=0 implies normal style of lines. style=1 implies dashed lines with segment and gap length of about 3mm. Any other value will result in undefined behaviour.

It must be kept in mind that all these modes, as well as any others that may be added, only come into effect when display takes place. The modes apply to any picture which is displayed. To cause a picture to be plotted the following procedure is to be used:

procedure plot (pic pict)

"pict" is plotted according to the display modes;

D.6. Filing Pictures

There will exist mechanisms that allow a user to consign a picture to a file and later combine this "filed" picture with other "filed" or "normal" ones. These pictures are in every respect like any other. They may be used within the program in the same way. The reason for the facility is that it is advantageous, especially when space is at a premium, to be able to place a picture onto backing store, and remove the corresponding data structure from core.

To store a picture in a file the procedure "save" is used:

```
procedure save ( pic pict ; string filename )
    store the set of points and their connections in the file called
    "filename";
```

To create a picture which contains a reference to such a file the following procedure is used:

```
procedure restore ( string filename  -> pic )
    the value returned is a picture containing the information that the
    points etc. are on the file called "filename" ;
```

When a picture is plotted, and it contains a reference to such a file, the file is read and interpreted as a series of points in logical coordinates, some joined to their successors, others not. Thus the transformations will also affect a "filed" picture in the same way as any other. The mechanism is fully general, but various operating systems may impose some restrictions on

the names of the files and on their creation. There are various errors possible, involving protection, non existent files, etc. These are generally system dependent. All the system dependent features are grouped together in appendices 1 and 2.

D.7. Miscellaneous Facilities

procedure dot (real x, y, z -> pic)

the value of this procedure is a picture of a dot at the point (x,y,z). The reason for it is that a picture consisting of a single point has no visible image;

procedure polygon (int n -> pic)

the value is an n-sided regular polygon centred on the origin. The radius of the circumscribing circle is 1. The first and last points coincide with the point [1,0,0] ;

procedure new.frame

The purpose of this routine is to clear the display. Generally anything on the display gets deleted. The exact effect depends on the display device;

Watch this space for more miscellaneous facilities.

Appendix D

Appendix D.1 : How to Use Graffiti on Unix (Provisional)

Programs should be prepared using the normal editing facilities. Graffiti is used at the moment as follows:

G prog.S [optional parameters]

Invoke the Graffiti compiler on the file prog.S. This creates a file "prog.out" in the current directory.

Gbind [device] prog.out

Bind the code with one of the device libraries. Currently the libraries are TEK, SIG and ANA, corresponding to the Tektronix 4000 series, Sigma display controller and the Anadex DP9501 printer.

s b.prog.out

Invoke the S-algol interpreter to interpret the S-code. Various magic incantations may be necessary to connect the output stream of the program to the desired device. Contact your local guru to find out what the devices are called.

Appendix D.2 : System Dependent Actions

When the procedure "save" is used to place a picture on a file, several situations can occur:

a) the file does not exist - it is created with mode

-rw-r--r--.

b) the file exists and is writable on - the file is overwritten but the old mode is retained.

c) the file is not writable - an error message is produced and execution stops.

In any case it is possible for the directory concerned to lack the needed permission (search in any case, and write in case a). An error message will be produced and execution will terminate.

When the procedure "file" is executed, no checking takes place. But if a picture containing a reference to a file is plotted, and the file does not exist or is not readable or is not a filed picture produced by Graffiti, execution will terminate and an error will be reported.

Redirection of Input/Output

The shell symbols ">" , ">>" and "<" can be used with executable files produced by the Graffiti compiler. If graphical output is placed in a file in this way it can only be viewed on the particular graphics device for which it was intended. Of course, as no Graffiti program is actually executed when

such a file is later displayed, there is no possibility of interaction. The display is then a purely passive device.

Appendix D

References :

1. An Introduction to Programming With S-algol

A. J. Cole and R. Morrison

Cambridge University Press 1982

2. Principles of Interactive Computer Graphics

W. Newman and R. Sproull

McGraw - Hill 1979 (2nd Ed)

3. Graphical Output Methods and Their Relation To Display System Design

C. Prosser

M.Sc. Thesis

Glasgow University Dept of Comp. Sci. 1981

Appendix E: Some Measurements

Overheads

In the first part of this appendix we look at the overheads of using Graffiti. These can be divided into two categories, space and time. There is a basic space overhead associated with the Graffiti run time library. By comparing a null S-algol program with a null Graffiti program after binding with the library the difference is found to be around 6000 bytes. This represents around 10% of the total program space of the PDP11.

The time overheads are more pronounced, but again they are fixed and do not change with the size of the program. The compilation takes longer, mainly because of the prelude. The prelude takes around 12 seconds extra CPU time on the PDP11 to compile. The figure for a null S-algol program is 2.3 seconds, while a null Graffiti program takes 14.2 seconds*. These figures are repeated for the compilation of a trivial program **cube**, a slightly longer **ackermann** and more reasonable length **8queens**.

* All the timings are the averages of at least seven attempts run on a load-free PDP11/45.

Cube

```
      Graffiti Graphics System    page    1

    3 --  procedure cube( int x -> int )
    4 --  x * x * x
    5 --
    6 --  procedure writeout( ( int -> int ) b )
    7 --  write b( 3 )
    8 --
    9 --  writeout( cube )
   10 --  ?

**** Program Compiles ****
```

Ackermann

```
      Graffiti Graphics System    page    1

    3 --  write "Input two integers "
    4 --
    5 --  procedure ack( int m,n -> int )
    6 --  if m = 0 then n + 1 else
    7 --  if n = 0 then ack( m - 1,1 )
    8 --  else ack( m - 1,ack( m,n - 1 ) )
    9 --
   10 --  write ack( readi,readi ),"'n"
   11 --  ?

**** Program Compiles ****
```

8queens

```
   Graffiti Graphics System    page    1

 3 --  ! Program to place 8 queens on a chessboard
 4 --  let n := 0 ; let more := true ; let count := 0
 5 --  let pos = vector 1::8 of 0
 6 --
 7 --  procedure add.a.queen ; { n := n + 1 ; pos( n ) := 1 }
 8 --
 9 --  procedure alter
10 --  if pos( n ) = 8 then
11 1-  begin
12 --       n := n - 1
13 --       if n = 0 then more := false else alter
14 -1  end else pos( n ) := pos( n ) + 1
15 --
16 --  procedure cantake( int i,j -> bool )
17 --  ( pos( i ) = pos( j ) ) or
18 --   ( abs( pos( i ) - pos( j ) ) = abs( i - j ) )
19 --
20 --  procedure incheck( -> bool )
21 1-  begin
22 --       let check := false ; let i := 1
23 --       while i < n and ~check do
24 2-       begin
25 --            if cantake( i,n ) do check := true
26 --            i := i + 1
27 -2       end
28 --       check
29 -1  end
30 --
31 --  !************ Main Program ****************
32 --
33 --  while more and n <= 8 do
34 1-  begin
35 --       if n = 8 then alter else add.a.queen
36 --       while more and incheck do alter
37 --       if n = 8 do
38 2-       begin
39 --            write "A solution to the 8 queens problem is "
40 --            for i = 1 to 8 do write pos( i ) : 3
41 --            write "'n"
42 --            count := count + 1
43 -2       end
44 -1  end
45 --  if more then
46 1-  begin
47 --       write "The number of solutions to the ",
```

```
48 --                  " 8 queens problem is ",count : 4
49 -1  end else write "No solution exists'n"
50 --  ?

**** Program Compiles ****
```

Program	S compile	G compile
null.S	2.3	14.2
cube.S	3.0	15.1
acker.S	4.3	16.6
8queens.S	8.5	20.7

As can be seen the Graffiti compile time overhead is roughly the same, 12 seconds, no matter how long the program is. This is what one would expect, as the only extra work the the Graffiti compiler does is the compilation of the prelude. This overhead diminishes in importance as the size of program grows, because the proportion of total compile time it represents diminishes.

The second part of the time overhead is associated with the binding of the compiled Graffiti program with the run-time library. This step can be avoided altogether with short S-algol programs but is unavoidable with Graffiti. The basic overhead for binding a Graffiti program is about 13 seconds. This rises slowly with the size of the program to be bound.

The third overhead is the extra time the Graffiti run time system takes to set itself up as the bound program is started, and any tidying up that needs to be done. This was found to be so small as not to be

measurable.

The overhead of preparing a Graffiti program is thus about 25 seconds. This is just about tolerable for a small program. As the program size increases thus the overhead becomes less and less noticeable.

Performance

The speed of a running Graffiti program is nothing spectacular. It does manage to keep up with the Tektronix 4662 pen plotter at 1200 baud, but cannot drive faster devices at their full speed. Nevertheless this seems perfectly adequate for most serial graphics devices. To produce a level 3 hilbert curve (see the **hilbert** program) takes about 9 seconds, while a level 3 sierpinski curve takes about 5 seconds. The next levels take 35 and 20 seconds respectively.

Sierpinski Curve

```
   Graffiti Graphics System     page     1

 3 --
 4 --  let eight.turn          =       z.rot( -pi/4 )
 5 --  let quarter.turn        =       z.rot( -pi/2 )
 6 --  let half.turn           =       z.rot( -pi )
 7 --  let three.quarter.turn          =       z.rot( -3*pi/2 )
 8 --
 9 --  let origin = [0,0,0]^[0,0,0]
10 --
11 --  let reflect             =       scale( -1,-1,1 )
12 --
13 --  let head       =        [-1,1,0]
14 --  let tail       =        [ 1,1,0]
15 --  let end.of.pic          =       [ 1,-3,0 ]
16 --
17 --  forward half.corner    ( int -> pic )
18 --  forward corner                 ( int -> pic )
19 --
20 --  procedure sierpinski ( int compl -> pic )
21 1-  begin
22 --        let shift := 2
23 --        if compl < 1 then shift := 4 else
24 --                for i=1 to compl do shift := shift*2
25 --        let c :=
26 --        ( if compl <= 1
27 --        then corner( 1 ) # move( shift,0,0 )
28 --        else corner( compl ) # move( shift,0,0 ) )
29 --
30 --        ( c#three.quarter.turn ^ c#half.turn ^
31 --          c#quarter.turn ^ c ^ end.of.pic ) # eight.turn
32 --
33 -1  end ! sierpinski
34 --
35 --  procedure corner ( int compl -> pic )
36 1-  begin
37 --        let HC :=
38 --        ( if compl<=1
39 --          then  half.corner( 1 )
40 --          else  half.corner( compl ) )
41 --
42 --        HC ^ HC#reflect
43 --
44 -1  end ! corner
45 --
46 --  procedure half.corner ( int compl -> pic )
47 1-  begin
```

```
48 --
49 --        if compl<=1 then        head ^ tail
50 --        else
51 2-        begin
52 --                let HC = half.corner( compl-1 )
53 --                let shift := 1
54 --                for i=1 to compl do shift := shift*2
55 --                let left  = move( -shift, 0, 0 )
56 --                let right = move(  shift, 0, 0 )
57 --                let up    = move(  0, shift, 0 )
58 --
59 --                HC#left ^
60 --                corner( compl-1 )#quarter.turn#up ^
61 --                HC#right
62 -2        end
63 --
64 -1  end ! half.corner
65 --
66 --
67 --
68 --  set.new.frame( false )
69 --  set.window.border( true )
70 --
71 --  while true do
72 1-  begin
73 --        let i = readi ; let throwaway = read
74 --        let size:=4
75 --        for j=1 to i do size:=size*2
76 --        size := size+1
77 --        set.limits( -size,size, -size,size, -1,1 )
78 --        set.window( -size+0.5,size-0.5, -size+0.5,size-0.5 )
79 --
80 --        plot( origin & sierpinski( i ) )
81 -1  end
82 --  ?
```

**** Program Compiles ****

Hilbert Curve

```
  Graffiti Graphics System    page    1

  3 --
  4 --  procedure hilbert( pic p ; real width -> pic )
  5 1-  begin
  6 --        let p1 = p # mat.mult( scale(1,-1,1), z.rot(-pi/2))
  7 --
  8 --        let p2 = p # mat.mult( scale(-1,1,1), z.rot(-pi/2))
  9 --        (p2 # move(width+1, width+1, 0) ^
 10 --         p # move( 0, width+1, 0) ^
 11 --         p ^
 12 --         p1 # move( 2*width+1, width, 0))
 13 -1  end ! hilbert
 14 --
 15 --  while true do
 16 1-  begin
 17 --  let complexity =readi
 18 --  g.out := s.o
 19 2-  begin
 20 --        let curve := [0,0,0] ; let width := 0
 21 --
 22 --        for i=1 to complexity do
 23 3-        begin
 24 --                curve := hilbert( curve, width )
 25 --                width := width*2 + 1
 26 -3        end
 27 --
 28 --        set.limits( -1, width+1, -1, width+1, -1, 1 )
 29 --        set.window(-0.5,width+0.5, -0.5,width+0.5)
 30 --        let origin = [-0.1,0,0]^[0.1,0,0] &
 31 --                              [0,-0.1,0]^[0,0.1,0]
 32 --        set.window.border( true )
 33 --        set.frame.border( false )
 34 --        set.line.style( 0 )
 35 --        set.new.frame( false )
 36 --        plot( curve )
 37 -2  end
 38 --
 39 -1  end
 40 --  ?

**** Program Compiles ****
```

Since the programs are very compute bound the actual wait times are only longer by a few seconds on an otherwise load-free system. If the system has to support other users the wait times increase accordingly.

The main reason for the relative slowness of Graffiti is the large overhead involved in the multiplication of matrices. Most of this is due to address calculation rather than arithmetic. A simple experiment disclosed the fact that if the matrix multiplication routine did not actually do anything except return an identity matrix, the speed of the program doubled. The matrices consist of four rows of four elements each and considerable improvement in speed could be achieved with a purpose built piece of hardware. It is also probable that a big improvement can be achieved by implementing some basic matrix (or just row) operations in the S machine. I am reluctant to do this, as it would then mean a different interpreter for Graffiti to that for S-algol.