Cosc 460 Project Report

Department of Computer Science

University of Canterbury

LAGS
Low Cost Animated Graphics System

Supervisor: Dr. R.E.M. Cooper

C. L. Williams

October 1985

Table of Contents

Acknowledgements

I would like to thank Buisiness Computers Ltd, and in particular John Kelly, for their support provided in the initial stages of this project, and for the use of the digital VT241 graphics terminal.

I would also like to thank my supervisor, Dick Cooper, for arranging this project, and for the help provided in the preparation of this report.

1. Introduction

The aim of this project is to look at the development of a "**Low cost animated graphics system**".

An animated graphics system is a system whereby graphic objects or scenes can be transformed over time so as to appear continuous or *realistic*. Such transformations can be expressed as from one place to another, or from one object into another etc.. An example of such would be cartoon animation on television.

A complete animated graphics system is one which will allow the user to generate realistic animation sequences with (hopefully) relative ease.

There are a few such systems in existence, but they are generally quite difficult to use and understand. Presuming that good quality graphics animation is required, they are also extremely expensive to acquire.

Most present animation systems provide the facility to view an animation sequence made up of frames. Each of these frames is made individually by the user making minor changes to the previous frame. This project is to *explore* the possibility of a different approach whereby many such frames can be generated by the computer on the users command, rather than painstakingly one at a time.

This approach is discussed in more detail, along with the problems associated with it in Sections 2. The possible solution methods and how each should be approached is covered in Section 3. Section 4 outlines the solution chosen, while Section 5 covers the implementation aspects of this solution. Because this project is very open-ended, the system itself allows many future developments, some of which are discussed in Section 6.

2. The Problem

Business Computers Ltd (BCL), a Christchurch software firm felt that an animation system was necessary which

1. allowed the user to animate objects without the need for the user to construct the objects in the first place; and
2. allowed animation sequences to be generated in an easier way than a frame at a time as discussed in the Introduction.

The design of the animated graphics system was proposed in the following manner :-

A central site would perform development work and creation of commonly used objects (standard objects) as well as any objects specifically requested. This *database* of objects could then be supplied to the users of the system.

The users would then create their own animation sequences using an *end user manipulation language* in conjunction with the objects supplied by the central site and those created by themselves. The resulting animation sequences would then be able to be recorded on film or videotape via an appropriate interface.

A typical user would be an advertising agency, but there are many other possible applications, such as television educational programs.

This approach appears to meet the objective of providing an animation system that is easy to use, provides objects for the user so that the user need not create their own, and provides the possibility of generating a complete animation sequence without having to build each frame individually.

This project has taken as its initial step to develop the appropriate end user manipulation language, which will now be referred to as LAGS -low cost animated graphics system language.

Depending on its context, LAGS can refer to the complete system or just the language.

2.1 Constraints

A typical advertising agency (or other end users) would require the following criteria to be met :-

2.1.1 Cost Effectiveness

The system must be cost effective. After consultation with BCL, a system is considered cost effective if it costs under $100,000.

There exists such systems at either end of the cost scale, but nothing in between. At the costly end of the scale would be expensive systems such as the Bosch FGS4000 costing $750,000. This particular system consists of a dedicated high resolution graphics computer that has a huge memory and many hardware driven graphics animation devices. At the other end of the scale would be graphics capable micros (such as an Apple III). This system, although cheap, requires the user to specifically code a particular animation sequence rather than utilise what this project is developing. Such a system would not be able to meet storage requirements that may be required by this system being developed. So one is far too expensive, while the other is cheap and unable to meet the requirements of an animation system.

To be able to use LAGS at the end user sites, two possible solutions are apparent :-

1. Each user has their own computer, and LAGS is implemented on such.
2. Each user is linked to a bureau (or more likely the central site) which provides a centralised computing facility.

After discussing both possibilities with BCL, and considering what the typical users are likely to prefer, it was felt that option 2 would not be satisfactory since

a) The reliability of one central site compared to separate user sites is obviously not as good. If the central site goes down, then all user sites are inoperative.
b) Timesharing systems are not particularly good for comprehensive graphics (in real time) since you can be timesliced out, and so the animation may pause at frequent intervals.
c) Unwanted communication costs in communicating with

the central site would occur since typical users are likely to be spread out over some area (or even the complete country!).

With using their own individual computers, the users would have none of these problems, and some could even use their existing hardware.

2.1.2 Portability

Given that the users each have their own installation, the question of what type of hardware should be used arises. Obviously, an advertising agency would not require, and could probably not afford, a mainframe system, but more likely a semi-dedicated work station or small system; this is most likely to be a larger micro or a very small mini.

If the users do each have their own system, what type of systems should be permitted?. LAGS could be designed with one hardware configuration in mind, whereby each user must have the appropriate equipment. This is not very reasonable on the end users, especially if they have equipment which may be suitable but does not meet that specified. It is therefore better to develop LAGS with a high degree of portability so that if the situation arose with a variety of computer systems requiring the use of LAGS, it is possible to do so. So, LAGS should be portable in terms of the machine, the language written in, and the graphics terminal used.

2.1.3 System Use

Since the system will be required to be show some form of *real time animation*, and that this is not really feasible for complex object/animation sequences, the system should ideally work in two ways

a) Interactively, to try out animation sequences in, say a lower quality form of output.

b) Via a program to produce the final animation sequence in *perfect format*. This may or may not be in *real time* since the viewing speed is limited by the computer, the graphics terminal, and the communication speed between the two.

2.2 Language Objectives

What should such a system be able to do?.

The system should be powerful enough to respond to user commands such as "place object at ...",
"transfer object to ... using movement type ...",
"transform object1 into object2 over ... secs/frames";
as well as handle more primitive commands such as "draw a line", "circle".

Most of the objects that the user of the system requires should be *predefined* objects stored in a database.
The user should also be able to easily make ones own objects, which can be in turn stored in the database. This would most likely be done by using primitive graphic commands as well as utilising existing database objects.

Many objects should be able to be animated simultaneously, although it is envisaged that only one object at a time will be achieved within the duration of this project. It is hoped that later development will allow for a scheduling algorithm to achieve this.

For this project, only 2 dimensional figures and transformations will be dealt with. It is intended that once the basic LAGS system has been developed, it should be possible to extend the system to handle such things as colours, shading, and 3 dimensions -this altogether would be too much to put in one honours project.

It is also hoped that it will be possible for text to appear in an animated form as if written or printed in the same way as if done by hand. A provision for this should be made in the command language.

2.3 Batch or Interactive

The question to decide here is to whether the animated graphics system should be batch or interactive processing orientated.

The disadvantage of Batch processing is that it would take a long time between formulating a graphics animation sequence, and actually seeing the results. This is not really suitable for the sort of system required.

The advantage of interactive processing is that the results of a command or sequence of commands can be seen almost immediately. The disadvantage of this is that as the complexity of the objects being animated increases, the time to animate increases to possibly an unacceptable level.

So interactive processing appears to be the best option. The only disadvantage is its speed.

An interactive system could be designed in one of two distinct ways.

1. The system could interpret commands and produce high quality output codes to a file, which could be subsequently displayed at maximum possible speed (still limited by hardware capabilities).
2. The system could respond immediately to each command, which may or may not be in a lower quality form of graphical output.

The solution chosen for LAGS is to use an interactive system that will respond reasonably quickly, but with the ability to generate a file or something similar containing a trace of the graphics done, which can then be used (or simply listed) to see the graphics as quickly as the terminal can handle.

3. Possible Alternatives

There exists four possible alternative solutions to the system as a whole, some of which have been tried in the past.

1. Use an existing graphics animation package.

2. Build a command language on top of an existing graphics package.

3. Extend an existing programming language to handle graphics and animation commands,
   either by a) library procedures or
   b) pre-processing.

4. Develop an entirely new language.

3.1 Use an existing graphics animation package

An advantage of using an existing animation graphics package is that as most of the animation would be done by the package, the only work would be to implement such a package onto the particular computer, and then investigate making the package portable. Unfortunately, no existing package, in terms of cost & size of computer required is known to exist!

3.2 On top of an existing graphics package

An existing graphics package would supply the basic object representation, primitive drawing and basic graphics commands. This is a definite advantage. With these features provided, only the actual animation commands, and the necessary non-graphic commands would have to be designed.

The disadvantages of such a package is that LAGS would have to be interfaced to the package, and the package would have to be interfaced to the particular hardware being used.
The existing graphics package may

1. be very large in size, and so could cause problems

with suitable end-user machines and portability.
2. place constraints on what animation commands can be done in terms of the command language to be built on top.
3. be inefficient since it would essentially be one package on top of another.

Example: Unigrafix. This is being used by Mr J.R.A. Collier to produce an animation system in association with Television New Zealand.

3.3 Extend an existing programming language

The advantage of this alternative is that all control and data structures that may be required in the animation language are present, so only the library procedures or preprocessing templates and preprocessor would have to be developed.

The disadvantages that occur with this solution are :-

1. A lack of flexibility is enforced by being constrained to the existing language's features.
2. It could be hard to read and debug the program.
3. It can't be made interactive, unless built on an interactive language such as Lisp, which are typically not very portable.
4. It may not be very portable. The likelihood of compiling LAGS for a specific machine would be remote if it were written in such a language.
5. It may not be possible to implement "easy and understandable" procedures or templates which perform complex animation sequences easily.

Example: [1] YACAS by T.J. Britton, - Algol60 extended by procedures.

3.4 Develop an entirely new language

By developing an entirely new language to handle animation, it is possible to provide for portability, in terms of both the computer and the graphics terminal required. There is also the freedom to make and expand the language to suit the animation techniques available and required.

The language can be made suitable for both interactive & non-interactive applications.

Apart from the fact that more work is required to implement a new language, the other main disadvantage is that eventually features such as scheduling, control & data structures that existing languages already have would need to be implemented.

4. The Solution Chosen

To Develop an Entirely New Language

The reasons for deciding upon this solution are that

1. A new language can be developed without any restrictions that a package or another programming language may cause.

2. The language can be designed in such a way that hardware portability will be attained. Portability for the system itself can be achieved by writing the system in a suitable language. Terminal compatibility, and hence portability, can be achieved by having a small "central core" of terminal-dependent graphics primitives, whereby the system only interacts with the terminal via these.

3. Concise commands can be produced which perform complex animation transformations while making the user interface understandable. Such commands can also be designed with storage and efficiency best utilised, whereas on a system that already provided graphics capabilities this may not be possible because of restrictions imposed by the underlying package.

4. A new language can utilise all the best features that other solutions offer, while avoiding any of the restrictions imposed by such.

5. A system based on a new language can be written in such a way as to ensure that the system can be expanded to provide new commands without too much modification.

6. Finally, an important reason from the user point of view, is that the language can be made user friendly, readable, and easy to understand and debug. This is not as straightforward with some of the other alternative solutions.

4.1 Outline of Solution

The LAGS system is composed two main elements. A database which contains four types of entities, discussed below, and a set of commands that operate on this database to produce graphic objects and animation sequences.

4.1.1 THE DATABASE

Four main types of entities exist in the database, which are as listed below.

1. Objects. These are the actual object descriptions that are defined by the user and used in animation sequences.

2. Functions. Functions are both system and user defined arithmetic expressions that can use system as well as other user defined functions. The functions are unary in terms of 'x'. These can be used to define the shape of a curve or a path that an object moves along.

3. Points. These are a sequence of coordinates which are used in the definition of objects as well as defining the path an object will move through while being animated.

4. Formats. These are used to describe text representation, in terms of height, thickness, written or printed and so on. This has not yet been implemented.

4.1.2 THE COMMANDS

The commands provided by LAGS are described in plain English in the following three sections. For a detailed discussion of a particular command's syntax, or further explanation of what it actually does, refer to the Users Manual - Appendix D.

4.1.2.1 Non Graphic Commands

A non graphic command is one that does not produce any actual graphics itself. They are simply commands that allow the user to add and remove entities to and from the system, to enquire as to what entities LAGS knows about, and *normal system* commands including such operations as reading from a file or quitting the session.

Remove an entity.

This command is used to remove any of the four types of database entities from the database. It can also used to remove a given object from the screen, but this is discussed in the next section.

Quit a session.

This command is used to quit a session of LAGS.

It is also used to signify the end of the definition of an object.

Undo the previous command.

This command is not implemented yet, but will undo the effect of the previous command.

Index all or a selection of entities.

This command either displays an index of one of the entity types if requested, or simply displays an index of all the entities known to LAGS.

Display an entity's details.

This command will display the definition of an entity, in the way that LAGS has interpreted it. This may not be in exactly the same format as the user has entered such.

Edit an entity.

This command calls the underlying system editor to enable the editing of the definition of an entity. The entity is actually removed from the database and passed to the editor in a format similar to that produced by the display command. It is then re-entered into the database upon the end of the edit session.

Help command.

This command displays the syntax and information regarding a particular command (if specified) similar to that in the User Manual. If no particular information is required, then a listing of all the commands available is displayed.

This command currently has the relevant information coded into the LAGS program - an improvement to allow easier updating of such information would be to retrieve this information from a file.

Equate object as an amalgamation of other objects.

This command, although not implemented yet, is intended to allow the definition of an object in terms of other objects, and so form a hierarchical object definition. The individual parts of this object should then be able to be accessed if need be.

Define entity command.

This command permits the definition of any of the four types of entities in the database.

Since the definition of an <u>object</u> by this command is likely to occur over a number of input lines, the LAGS prompt changes to remind the user what is being done - the last command in an object definition must be the quit command.

The definition of a format entity is not yet implemented in LAGS.

Load file.

This command opens the given file for subsequent use by the Run command.

It is intended to allow the file to contain either LAGS commands created by the user, or a storage efficient representation created by LAGS itself. The former case is only implemented at this stage.

Run file.

This command causes subsequent commands to be executed from the given file (or the currently opened file if no file is specified). Once all commands have been executed the input switches back to the terminal.

Save file.

It is intended that this command will save the current status of LAGS, for loading via the Load command. This command is not yet implemented.

Set command.

This command allows the setting of the screen coordinate system.

It can also be used to initialise and terminate three types of traces, which can go to either the terminal or a file. One traces the commands entered during the session, another the graphics codes generated by LAGS, and finally a dump trace showing how LAGS has interpreted the commands themselves. The latter trace is only of use for debugging LAGS itself, and is no help to the user.

System command.

This command allows a system call on the underlying operating system. Upon completion, control passes back to LAGS.

Where command.

This command displays the current cursor coordinates and the corner coordinates defining the screen.

4.1.2.2 Simple Graphics Commands

Simple graphics commands are commands that produce *simple* graphic results; they do not generate complex curves or any animation sequences or such like. They can be thought of as the primitive commands of an animation system.

Place an object at a point.

This command allows the placing of an object on the screen, and can optionally be used to name that instance of the object as something else. This is so that if any transformations are done on the object, the original *blueprint* can stay intact.

Remove an object from the screen.

This command is the same as discussed in the non graphics commands, only here will remove an object from the screen. At present a "trace" of where the object had been is left behind, if an object was behind it (the same occurs in an animation sequence as the object moves along). This is because an object is presently removed by redrawing it in inverse mode, so the background object would not be fully restored.

Print text at a point.

This command is intended to allow for the printing of text according to some specified format entity. At present, formats are not implemented, so this command only prints the text itself where required.

Fill an object with a colour.

This command allows for the filling of an object with a specified colour (or part of an object in the case of the intended hierarchical structured objects).

At present this command is not implemented, although LAGS will recognise its syntax.

Move to a point (either relative or absolute).

This command allows the movement of the graphics cursor to the coordinate specified, which can be absolute of relative to the current cursor position.

Plot at a point (either relative or absolute).

This command works in the same way as for the move command, except that it plots a point at the new cursor position. If no point is specified, then it plots a point at the current position.

Draw a line to a point (either relative or absolute).

This command, in its simplest form, draws a straight line from the current position to that specified. For a more powerful form of the command, refer to the next section.

Circle drawn at a point with a given radius.

This command draws a circle at the specified point, or the current position if none is specified, with a radius as specified.

Ellipse drawn at a point with given radii.

This command draws an ellipse at the specified point, or the current position if none is specified, with the maximum and minimum radii as specified.

Arc command.

This command allows the drawing of arcs in two forms, both starting at the current point. One is about a given point by a specified degree, while the other is through one point to the ending point.

Clear screen.

This command clears the screen and notes any objects that were on the screen as being removed.

4.1.2.3 Complex Graphic & Animation Commands

These commands are the most powerful and complex that LAGS provides to the user. All but one are the heart of the system as a whole - the animation commands.

Draw to a point (either relative or absolute) using points.
This command allows plotting through a sequence of coordinates, either curved or in straight segments. The points themselves can be either points as from the points entity of the database, or function generated points.
If a curve is to be generated, LAGS uses the inbuilt curve fitting of the VT241 terminal in the present implementation.

Transform command.
It is intended that this command will transform one object into another over a number of frames. For example, it could be used to transform a square into a triangle.
This command is not yet implemented.

Rotate command.
This command will rotate an object about a point (or its centre) by a given degree over a number of frames. An extra facility is provided for this command which is discussed below.

Scale command.
This command will scale an object about a point (or its centre) by specified scale factor(s) over a number of frames. An extra facility is provided for this command which is discussed below.

Transfer command.
This command will transfer (move) an object to a point (either relative or absolute) along a path over a number of frames. The path that the object is moved along can be a straight line from the start to the end point, or it can be along a path defined by a coordinate sequence. The coordinate sequence (or points) are the same as those used in the complex draw command in this section; that is they can be points as in a points entity, or function-generated points. An extra facility is provided for this command which is discussed below.

The above three commands (rotate, scale, & transfer) will also allow the occurrence of simultaneous rotations and scalings about specified points (either fixed or relative to the object -relative means that the point moves in relation to the object). A seemingly infinite number of such transformations could happen to an object at once.

Most of the commands don't actually require the specification of a number of parameters - for example, by not specifying a coordinate, the current position is assumed; by not specifying an object, the last one used (ie the current object) is assumed.

5. Implementation of LAGS

LAGS has been implemented on the Vax by an interpreter written in 'C'. A digital VT241 Graphics terminal is used under the ReGIS graphics system, which has kindly been lent by BCL.

5.1 Why Interpreter?.

A compiler would be used to compile a LAGS command language program into object code, whereby it could be linked with the appropriate libraries and executed. The compiled code from a program would most likely contain a number of calls to modules in a LAGS utility library.

The main reasons for basing LAGS on an interpreter rather than a compiler are that

1. An interpreter can be developed faster and more easily than a compiler.
2. A compiler would not be portable unless it produced intermediate code (eg EM-code).
3. An interpreter allows commands to be modified more easily.
4. If an interpreter is written initially, it makes the job of producing a compiler (at a later date) easier, since the chance of modifications for the compiler would be reduced. This is an advantage as it is inherently more difficult to alter a compiler than an interpreter given the case of a modification to the command language.

If a compiler was used, the main advantage would be the speed improvement over an interpreter in the execution of the commands.

It was decided that since an interpreter can be developed more quickly, it would be the best place to start. A trace facility could be added to give even better speed than that for compiled if required.

5.2 Why written in 'C' ?.

The language to write the interpreter had to be portable, and had to be on an available computer. Since the *best* machine appeared to be the Vax, which runs UNIX, the obvious first choice seemed to be 'C'.

Another advantage in using C, apart from its portability, is its lack of highly restricted structures, which allows easier programming (as long as you know what you are doing) and the production of probably more efficient code.

A later suggestion was as to why Lisp was not considered. After all, the advantages of Lisp was that it meet the interpreter requirement, and that it is possible to develop a new language on top of Lisp with relative ease. The major drawback regarding Lisp was the portability aspect, and its execution speed even when compiled is unlikely to be better when compared to a C based interpreter.

Once the interpreter was well on the way of being written, a further question was put as to why automatic parser and scanner generating tools were not used to *speed up* the development of the interpreter. Such tools exist for C programs, namely Yacc and Lex.

Upon looking into this possibility, the following disadvantages of using such tools for this project became apparent :-

1. The use of such generators requires the knowledge of the syntax and grammers that they require as input. It appears that it would have taken more time to learn how to use such grammers and apply this to the language LAGS than it took to actually write the corresponding sections of LAGS without such tools.
2. Updating the language of LAGS would be a disadvantage for anyone that did not know the grammers required as input since such would have to be learnt.
3. Portability would be a problem if LAGS was to be implemented onto a computer system for future development work that <u>did not support</u> such tools. This would mean the code produced by such tools would have to be edited, which would not be an easy task as the code itself is not very *readable*.

Thus the main advantage of using such tools is ease of modifications <u>once the grammer has been developed</u>. The main disadvantage mentioned is the portability of LAGS to new system for development work. This portability aspect is considered far more important, so it seems reasonable to not have used such tools at the time.

<u>5.3 Terminal Independence</u>

To achieve terminal independence, it was necessary to design a small central core of terminal dependent graphics primitives. All animation commands therefore had to communicate with the graphics terminal via these primitives.

One problem in developing these primitives was to determine what level of primitives should be used. For example, should a 'line drawing' primitive be provided, or a non primitive 'line drawer' which utilised the primitive 'plot' to plot the line.

The decision to determine the appropriate level of terminal independence became based on the following three points .....

1. How important is the level of primitives?, that is, would the time spent on developing 'very' low level primitives, (bearing in mind the extra work required for higher level routines to utilise these), be beneficial to the main objective of developing an <u>animation</u> system. The time may be spent more wisely on developing the animation techniques, within the time allocated for the project.
2. How *dumb* can a graphics terminal be expected to be?.
3. What effect on the *real time animation speed* will the level of primitives (& hence software layers) have?.

Considering points 2 & 3, it was decided that the graphics terminal should be *reasonably intelligent* whereby primitives for such things as plotting, line drawing, and screen clearing should be provided by it.

As the level or complexity of the commands or primitives increased, consideration 1 started to play some importance. Because of this, it was decided that circle and arc drawing commands should also be terminal independent (to produce code to do this at a later date would not require too much effort as appropriate formulas are well known).

Two higher level primitives that the particular terminal being used provided had to be considered, particularly from the point of view of the time they would actually require to implement in LAGS.

1. Inbuilt curve fitting. Typically only the most comprehensive graphics terminal would be able to do this - so should the time be spent in implementing a curve fitting algorithm?. Some of the known curve fitting techniques were looked into, predominantly B-splines, and it appeared possible that a suitable method could be used. However, for the time scale of the project, it was decided to use the inbuilt curve fitting provided by the terminal, with a recommendation that a suitable algorithm could be added in later developments. Implementation time required and execution speed were the main factors which affected this decision.

It should be noted that the inbuilt curve fitting is used in four different situations, namely curve fitting of points, curve fitting of function generated points, drawing an ellipse, and drawing an arc. For a discussion as to why the curve fitting was used, and how further developments may avoid using the inbuilt curve fitting, refer to the Section on 6.1.

2. The other built-in feature of the terminal considered was the ability to store a sequence of graphic commands defining an object at the terminal - this is basically a macro. This would allow quicker redrawing of the object as only one command would be required.

The reason against utilising this feature is that the object has to be stored within LAGS itself for obvious reasons, and since the redrawing of objects is not very slow, it seemed pointless to use this feature; which would have to be removed eventually for portability reasons.

5.4 The Animation

The mathematical basis explaining basic graphics transformations is covered in Appendix A.

The results of this is that the actual animation is based on ten constants which are pre-calculated before a given animation sequence starts. It is then only a matter of performing the necessary operations involving these constants to transform an object frame by frame. The effect of this is that once the constants are calculated, the animation flows as smoothly as possible.

An object is animated by the modification of the parameters of the commands defining the object, which are easily accessed by the "list of pointers to numbers" accompanying each object definition. (This is explained within Section 5.6.1). Thus the actual updating of the object is simplified greatly as there is no need to actually scan the object and determine what commands it consists of, and then proceed to alter the parameters.

5.5 Screen Representation

The screen is divided into two windows. One is a text window which is at the bottom of the screen. The other consists of the rest of the screen and is known as the graphics window.

In the current implementation, the text window is defined via a terminal dependent primitive, which reserves the bottom 5 lines of the screen for this use. The terminal handles all the necessary scrolling within this window. This may not be satisfactory for some terminals, which may cause software routines handling such to be written for LAGS.

The graphics window actually covers the whole screen (including the text window!) in the current implementation. To achieve the effect of a separate window, the appropriate offset is added to all 'y-axis' coordinates to effectively shift the graphics region up above the text window. This again is terminal dependent.

Commands such as Index, Display, and Help currently use the reduced text window, which is not satisfactory as all the information produced by these commands cannot be seen at once. Some method is required so that all the information is seen in response to any of the above commands. One way would be to temporarily enlarge the text window to fit all the information, and then reconstruct the screen afterwards. Another way would be to allow a text window to be shown at a time, with the user signalling when to proceed.

A similar problem happens with the Edit command where it would be preferable if the system could switch from the graphics window to a full sized text window.

One way that LAGS could solve both problems would be to record all the graphics codes produced that are required to reconstruct the screen (since the last clear screen). Then it could do the screen changing itself. The Set Graphics Trace Command, which is executed within the primitives, could also send to "screen recording" a copy of the trace to keep track of what is needed to reconstruct the screen. Other more elegant and more efficient ways may be possible.

Another feature of screen representation which is not carried out by LAGS itself is that of clipping. Clipping is the term used to describe how coordinates going to the screen are checked for validity. If a point is not on the screen, the following can occur :-

- an error occurs;
- the point is *wrapped around* so that it appears somewhere where it shouldn't;
- the point is not displayed.

The last option is what clipping tries to achieve, so that the system does not have to rely on the terminal itself. On the current implementation the terminal itself does clipping *most* of the time.

There are many clipping algorithms known, from basic ones such as checking every point, to more complex and less time consuming methods. Of course, whatever is chosen, an extra amount of overhead will occur, which will of course reduce the *real time speed* of LAGS.

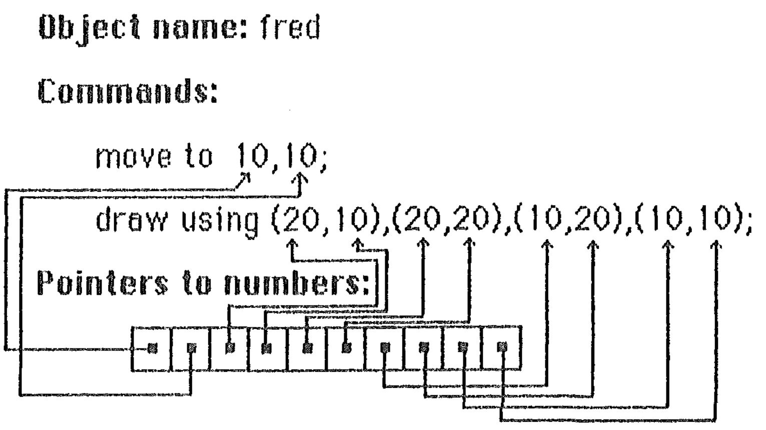
5.6 Database Entities Representations

The four main entities (objects, functions, points and formats) of the database and what their uses are have been outlined in Section 4.2.1. All these entities can be referred to by the commands of the language in two ways, namely an identifier, or a number (which is what the entity is known internally to LAGS by, but is available to the user).

This section now covers briefly how each of these has been implemented.

5.6.1 Object Representation

An object is stored as a sequence of commands which actually draw the object. An additional structure is required within the object's definition that points to all numbers *enclosed within the commands* which define the object. This is so that any animation is able to be carried out more efficiently. As to why this is so, refer to the Section 5.4. A diagram showing this structure is as follows, whereby to alter the parameters of the commands making up the object, one only has to scan the Pointers to numbers.



This idea could even be extended further whereby the object is represented as a string of graphic codes (terminal dependent) and the "pointers to numbers" point to the numbers to be transmitted within this string. This would mean that an object would only have to have its defining commands executed once, unlike the current animation where such must occur at each frame.

Because of the animation aspect, that is parameters of the object are actually altered during an animation, an object must be stored in terms of absolute or actual coordinates, and not relative. This means that at the point an object is actually defined, any commands referring relative to another coordinate must be converted to an absolute coordinate based command. For example, how would scaling by different amounts in each axis direction effect the following instances ? :-

a) a degree for an arc command.
b) a radius for a circle.
c) a relative coordinate, eg move by 10,20.

None of these could be treated in the same manner as absolute coordinates.

5.6.2 Function Representation

Functions are defined as a normal arithmetic expression, which can utilise other user defined functions or system supplied functions (eg sin, sqrt etc). The only functions that the user can define are unary, in terms of 'x'. No doubt, when 3 dimensions are to be implemented, then functions using two variables may be required. For details of the types of operands recognised, refer to the User Manual.

The expressions forming the functions had to be stored in such a way so that LAGS could use the functions efficiently. Because of this, all such expressions are stored in Reverse Polish Notation.

5.6.3 Points Representation

Points are recognised by LAGS as a finite sequence of coordinates. These can become known to LAGS by the "define points" command, and by the implicit use of coordinate sequences (the actual use of coordinates within a command, rather than a points identifier). In the latter case, LAGS assigns an internal points number to the sequence, and treats it the same as any other sequence from then on.

LAGS also generates temporary points that have been generated by the use of a function.

Points are stored as a linked list of coordinates.

5.6.4 Format Representation

A format is to be used by LAGS to specify the format specification used via the print command for displaying text in various forms.

This feature has not been implemented in LAGS yet.

5.6.5 Storage Structures

These four entity types have each been implemented with their own table. An entry in the table can indicate whether the definition of the entity has been loaded into LAGS (or still in the non-implemented database). When a new entity is to be added to a table, the table is scanned to find the next free space. The position an entity occurs in its table is known as the entity's number, and can be referred to by the user. The points table has twice as many elements to permit LAGS to generate its own (temporary) points entities without reducing the users' number.

Another table stores the identifier names of all the entities. It is implemented as a hash table with chaining to resolve collisions.

5.7 Path Representation

It should be noted that the technique discussed here to define a path is the same as that used to draw a curve or a sequence of line segments as done by the Draw command.

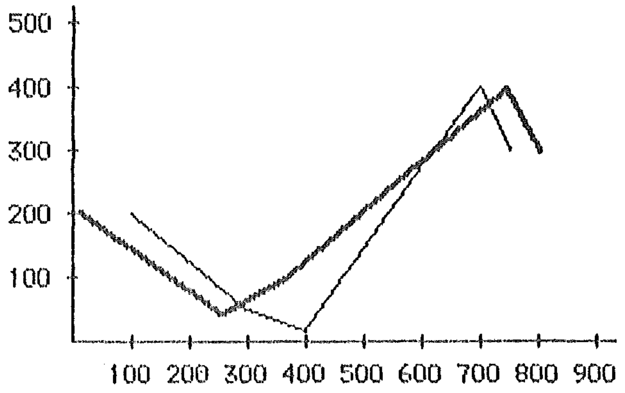
As has been explained, the path that an object moves along while being animated can be one of three types :-

1. a direct path from the start to the end coordinates (straight line).
2. a path following a coordinate sequence as specified by a points entity.
3. a path following a coordinate sequence generated by calls to a function entity.

There is no need to discuss case 1, as it is straightforward. In the case of a path specified by a function, LAGS generates a coordinate sequence by calling the function over the range of x_coordinate values, in appropriate steps, as specified within a command. Consequently this case can be treated identically to case 2.

LAGS knows the start and end coordinates of where the object is to go. It also knows the start and end coordinates of the coordinate sequence that is to be used to define the path..These points are typically not the same, so LAGS *stretches* the coordinates of the coordinate sequence so that the two start and end positions are the same. The mathematics involved in doing this is covered in Appendix C.

An example of such is shown in the following diagram.



— original points (100,200),(300,50),(400,10),(700,400),(750,300).

— new points generated by stretching to match start (10,20) and end (800,300). giving (10,200),(253,50),(374,100),(739,400),(800,300).

Once the path is defined, an animation along such a path first calculates the length of the path and then apportions the animation frames equally along the path according to this distance.

5.8 Summary

The LAGS system that has been implemented on the Vax does all that has been stated.

One can define objects, points and function entities, although they do not at this stage go into any database. Consequently the user can create a database by loading such definitions in a file and using the Run command in LAGS.

An object can be defined in terms of the following LAGS commands; Arc, Draw, Move, Plot, Print, Circle, and Ellipse. This permits complex objects to be defined, although they are not solid or hierarchical at this stage.

The user can perform scalings, rotations and movements of objects along any path. As the command language states, the rotations and scalings can occur relative or fixed about a specified coordinate, and as many of these can occur at once as is required.

On the Vax, all such animation sequences are performed as quickly as the terminal can handle. There is of course an initial pause (up to 2 secs real time) while the constants for the animation are calculated, and then the animation proceeds swiftly (with the occasional pause as LAGS is timesliced).

Since the Graphics Trace facility works, an interesting test was carried out to see if the animation could be speeded up even more. A trace of an animation sequence was sent to a file. The baud rate between the Vax and the terminal was doubled to 19200 baud and LAGS was given a high priority (so it had nearly all the CPU). A listing was then done on the trace file (by the Unix command 'cat'), whereby the animation proceeded. The actual speed of such an animation was barely any different from that achieved by LAGS interpreting the commands itself! (remember that with the trace on LAGS is actually performing twice the output). From this the conclusion can be drawn that the graphics terminal itself appears to be the bottleneck of the system, and not LAGS or the Vax. That is the part of the system which limits the speed of the animation.

6. Future Developments

The are many possible areas that can be developed to meet the main objective of producing a complete animated graphics system.

Some of the more relevant and important ones are listed below :-

- system curve fitting.
- hierarchical definition and animation of objects.
- increasing the terminal independency.
- implementing text & format specifications of such.
- implementation of the database.
- new subsystem for defining an object.
- moving to 3 dimensions, shading and multiple light sources.
- allowing many objects to be animated at once.

Each of these points and likely solution(s) is now discussed in more detail where possible.

6.1 System Curve Fitting

In the present implementation of LAGS, the inbuilt curve fitting of the terminal itself is used.

This is not really satisfactory when considering terminal independence. The reasons why this was used is discussed in Section 5.3.

The curve fitting is used in four distinct cases, namely
curve fitting of points, curve fitting of function generated points, drawing of ellipses, and drawing of arcs.

The reasons for each of these, and possible solution(s) is discussed below.

Curve fitting of points

As discussed in the previous mentioned section, the solution to this is to use some curve fitting algorithm, and generate the points on the curve via this. It may be possible to do this in a similar manner to the way the current terminal does it by only requiring the previous two points and working off the next one (doing it this way, if it were possible, would mean that the primitives for curve drawing themselves would only have to be modified).

Curve fitting of function generated points

One way to solve this is to have the function generate points that are close enough to plot the curve directly from these points.

Another way would be to treat such points as just "points" and submit these to the curve fitting solution as above. This is what is currently done now since function generated points are used to both define a curve to be drawn, and to define the movement of an object (former requires more precision than the latter).

Drawing an Ellipse

An ellipse must be stored as absolute points, so the four outer points have been chosen; refer to the Section 5.6.1 as to why this is so. Curve fitting was used here because it worked perfectly for an ellipse, and it was efficient.

Two possible solutions exist (apart from just solving the first case and still using curve fitting), while still retaining the absolute points.

One is to use four arcs to draw an ellipse. A suitable formula was calculated before the curve fitting efficient possibility came along, and was only partially implemented. This can be found in Appendix B.

The other way to produce the ellipse would be to just plot all the points of the ellipse according to the equation of an ellipse.

Drawing an Arc

The reasons that curve fitting was used here rather than the inbuilt arc command was that is was easier. Remember that the arc, like the ellipse, must be stored as absolute points -three in this case.

This could be implemented by using the curve fitting discussed above, using a terminal dependent arc command, or simply plotting the points of the arc.

6.2 Hierarchical Definition and Animation of Objects

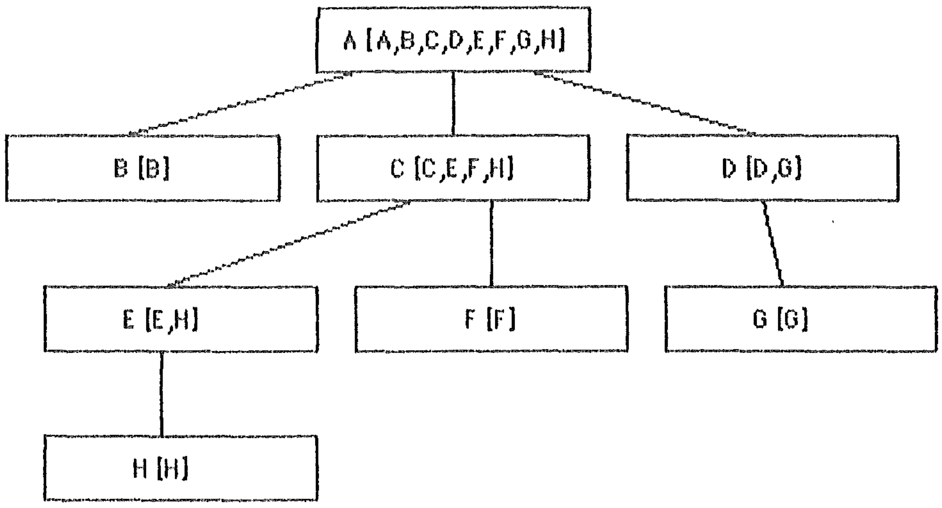
What is required here is the facility where an object can be defined as an amalgamation of other objects. An initial approach to the syntax for such a command has been done, which is described by the **equate** command as in the User Manual.

Given this possibility, the whole object or just parts of it could be referred to, particularly in the animation commands. This would add a great deal more power to the animating ability of LAGS. For instance, a complete object could be doing one thing while part(s) of it could be doing other things.

It is envisaged that once this structure was implemented, the subparts of the object could be accessed by *dot notation*. For example, "transfer person.arm.hand by 100,200".

The main animation commands could be easily modified to suit this, and the while clauses could allow the specification of subparts. For example, "transfer person by 100,100 while rotating arm.hand by 10deg and scaling arm by 2,2 over 10 frames".

To implement the hierarchical definition of an object within the object definition, one point must be remembered. To ensure that the commands work on all parts of the object as expected, the top level "list of pointers to number" (refer to Section 5.6.1) must point to <u>all</u> parts of the object, including subparts. A subpart will only point to all those points on its definition and those parts below it, but not above. This is shown in the following diagram.



Each part of the object is labelled A, B, ..., H. The section labelled 'A' is the top of the object hierarchy. The labels within brackets at each level is a list of those sections which the "pointers to numbers" of that level can point to.

With this in mind, the implementation of such into LAGS should not prove too difficult.

6.3 Multiple Object Animation

Currently, LAGS can only animate one object at a time. The ability to animate many objects simultaneously would be a requirement by any end user, and so it will have to be implemented.

To implement such, a scheduling algorithm would have to be devised. Apart from this a command structure would have to be developed so that LAGS recognises when this is meant to happen, but in a way that is still very understandable to the user.

The first approach to the solution would be to look at languages such as Simula Classes to see how such scheduling of events or processes is done. This in itself is straightforward, and could be easily applied to LAGS. For example, an agenda of animation instances could exist with which the animating routine selects the instance at the head of the agenda and performs one frame of animation. The resulting instance could then be placed at the end of the agenda. An instance recorded on the agenda would record such information as the object number, constants of the animation, the object starting position and so on.

6.4 Terminal Independency

To increase the terminal independency of LAGS, apart from the areas already discussed (such as curve fitting), the following should be done.

A number of the terminal dependent primitives should be written for a selection of commonly used graphics terminals. In doing so, it may be found that some of the primitives may have to be substantially altered, and not only at the primitive level. For example, the text windows setup may not be possible on some terminals.

6.5 Text Handling and Formatting

To achieve the type of text display and animation hoped for in the Section 2.2, a lot of development work would be required, especially in the area of getting text to appear in an animated form as if written by hand.

The amount of work required here, and its likely complexity, would be sufficient for a separate honours project.

6.6 DataBase Design

At the moment LAGS has the necessary functions called when access to the database is required, either loading of saving data. Unfortunately, these routines do nothing at the moment as there is no separate database.

To use commonly used entities at the moment, the only way to do so is to run a command file via the LAGS run command.

To implement such a database, the question of storage formats, access methods (what type of indexing) and so forth will be required.

The job to do this would be fairly time consuming, but should quite straightforward.

6.7 Shading, 3D, Colour, and Multiple Light Sources

All these are substantial sections to implement in themselves. Because LAGS has been written in a modular and consistent form, the job of implementing some of these extras should be simplified.

The next improvement to try would probably be moving LAGS into 3 dimensions, which would require the coordinate system to be changed from 2D to 3D. This would mainly effect the object drawing and animation effects, although the method used for the animation technique itself can be easily expanded to handle 3 dimensions (4x4 matrices would be used rather than the current 3x3).

6.8 Object Definition Subsystem

It is thought that the object definition could be changed so that a user can develop the object (possibly with the aid of a mouse) on the screen, rather than entering commands as now happens.

Two possible solutions as to how to achieve this would be :-

- a standard drawing package (say MacDraw for the Macintosh as an example) could be used to produce codes representing the object. An additional program would then convert these codes into something LAGS recognises which would probably be difficult to do.
- a specially written program which produces the correct object definition as expected by LAGS.

Many questions surround this improvement (- for example, who is going to use it, just the central site or the users as well), and all would have to be taken into account.

6.9 New System for Entering Commands

To enter commands to LAGS at the moment requires the knowledge of the language LAGS itself, and such commands must be typed in.

An improvement over this would be to provide a *mouse based menu* which could be used in a similar way to MacDraw for example, whereby the user would need no knowledge of the underlying command language.

To consider an improvement of this type a section of terminal dependent routines would be required to control the menu. The first step for this may be to develop such a subsystem on an Apple Macintosh since many of the tools for doing such are provided. Once it was found feasible on such a system, the same ideas could be transferred to other terminals.

6.10 The Improvements?

All these improvements covered, and there are probably more, will generate extra overhead on LAGS. A point may come where an improvement makes LAGS unacceptably slow or large, so some tradeoffs would have to be made.

7. Conclusion

This project has had as its main objective to determine the feasibility of developing an animation system that removes the need from the user to generate complex or commonly used objects for use in animation sequences, and to provide the facilities to allow easy creation of such sequences without the need to generate each frame individually. It has been shown that a suitable user friendly language can be developed to perform these tasks which the user can use.

This system, because of its experimental aspect, has enabled the possiblity for many interesting future developments.

The system as its stands at the moment can be developed into a powerful system with such features as three dimensions and simultaneously animated multiple hierarchical objects without too much difficult work. Extra features such as an object generation subsystem will of course enhance the appeal to prospective users.

References

[1] T. J. Britton.
YACAS - A Batch Computer Animation System.
Computer Science Master's Thesis,
University of Canterbury, 1978.

[2] W. M. Newman and R. F. Sproull.
Principles of Interactive Computer Graphics, 2nd Edition.
McGraw-Hill, 1979.

[3] Henry Fuchs (Editor).
Computer Graphics. SIGGRAPH '81 Conference Proceedings.
SIGGRAPH-ACM
Vol. 15, Number 3, August 1981.

Appendix A

Animation Mathematics

How is the animation achieved?

As per [2] and other references, the following matrix forms of transformations are known to transfer a point [x y 1] to [x` y` 1].

transfer by an amount (relative) (T_x, T_y)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -T_x & -T_y & 1 \end{bmatrix} \qquad \text{refer to as } T[x,y]$$

rotate about the origin by θ

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{refer to as } R[\theta]$$

scale about the origin by (S_x, S_y)

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{refer to as } S[x,y]$$

So, a rotation about a point other than the origin is achieved by simply transferring the point to the origin, performing the rotation, and then transferring back. This is shown as the following product

$$T[-c_x, -c_y] \cdot R[\theta] \cdot T[c_x, c_y] \qquad \text{refer to as } R[c_x, c_y, \theta]$$

which is a rotation about point (c_x, c_y) by θ.

Similarly for scaling giving $S[c_x, c_y, S_x, S_y]$.

Extending on this basic idea, a rotation by θ about a point (a,b) and also a scaling in relation to (e,f) by (s_x,s_y) can be done. This would give the final transformation matrix as

$$T[-a,-b] \cdot R[\theta] \cdot T[a,b] \cdot T[-e,-f] \cdot S[s_x,s_y] \cdot T[e,f]$$

or $$T[-a,-b] \cdot R[\theta] \cdot T[a-e,b-f] \cdot S[s_x,s_y] \cdot T[e,f]$$

These basic ideas have been extended to enable complex transformations to occur with relative ease whereby most calculations can be performed before the animation itself starts.

This is done by precalculating a composite matrix and using factors extracted from this in the subsequent animation.

Now, given $R[x, y, \theta']$ and $S[x, y, S_x, S_y]$ which allow rotation and scaling about a given point (x,y), the following has been formulated.

If an object (or whatever is being transformed) is moving in relation to its starting point, then two possible interpretations can be made of the rotation and scalings, which are

fixed: the point (x,y) remains fixed regardless of where the object goes to.

relative: the point (x,y) moves in relation to the centre of the object as it moves, so as to ensure that the point remains the same position in relation to where the object started.

If both these possibilities are allowed in the one animation sequence or command, it can be seen that an essentially *infinite* number of transformations could occur to an object at once, since either 'relative' or 'fixed' can occur in relation to <u>any</u> point.

So, along the lines of the above composite matrix calculation, it is possible to see how further multiplications of different transformations will actually produce two distinct composite matrices - one for 'fixed' transformations, and one for 'relative'. These two will now be referred to as FIXED and RELATIVE respectively.

If these matrices are to be repeatedly applied to the defining points of an object, and thus animate it, then FIXED is used just as it is. However, for RELATIVE to work, the object must be transferred back to where it started from, the relative transformation(s) performed, and then transferred back. So, with this in mind, the composite matrix becomes the product of

$$T[-T_x, -T_y] \cdot \text{RELATIVE} \cdot T[T_x, T_y] \cdot \text{FIXED} \qquad (1)$$

where T_x and T_y are the differences from where the object is now to where it started from (based on the objects centre).

Now since T_x and T_y can change as the animation proceeds, the above product cannot be calculated prior to actual animation time. Instead, the above computation must be simplified so that the transformation on the object can be done with little computation (compared to four matrices multiplied together, which then multiplies each point of the object).

This allows the following to be formulated, as on the next page.

Rewriting (1) as

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -T_x & -T_y & 1 \end{bmatrix} \cdot \begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix} \cdot \begin{bmatrix} g & j & 0 \\ h & k & 0 \\ i & l & 1 \end{bmatrix}$$

gives

$$\begin{bmatrix} ag + dh & aj + dk & 0 \\ bg + eh & bj + ek & 0 \\ \begin{array}{l} g(-T_x a - T_y b + c + Tx) + \\ h(-T_x d - T_y e + f + Ty) + i \end{array} & \begin{array}{l} j(-T_x a - T_y b + c + Tx) + \\ k(-T_x d - T_y e + f + Ty) + l \end{array} & 1 \end{bmatrix}$$

Now each point [x y 1] is multiplied by this matrix to get the new point [x` y` 1]. Treating the above matrix as

$$\begin{bmatrix} A & D & 0 \\ B & E & 0 \\ C & F & 1 \end{bmatrix}$$

gives

	$A = ag + dh$	$D = aj + dk$
	$B = bg + eh$	$E = bj + ek$
	$C = C1 + T_xC2 + T_yC3$	$F = F1 + T_xF2 + T_yF3$
where	$C1 = gc + hf + i$	$F1 = jc + kf + l$
	$C2 = g(1 - a) - hd$	$F2 = j(1 - a) - kd$
	$C3 = h(1 - e) - gb$	$F3 = k(1 - e) - jb$

It can be easily noticed that A, B, C1, C2, C3, D, E, F1, F2, F3 are all constant. This allows us to calculate the above constants <u>before</u> any animation starts. Given the ability to precalculate these constants, the new point [x` y` 1] can be easily calculated as

$$x` = Ax + By + C1 + T_xC2 + T_yC3$$
$$y` = Dx + Ey + F1 + T_xF2 + T_yF3$$

This means that each new object position can be efficiently calculated in the same speed no matter how complex the initial rotate/scale fixed/relative clauses were.

Appendix B

Ellipse Mathematics

Using Four Arcs to Draw an Ellipse

Given the four outside points, the two radii of an ellipse can be easily calculated -call x and y. Assuming that x > y in this case the following can be determined :-

(centrex,centrey) is the centre of the ellipse.

$b = \sqrt{(x^2 - y^2)}$;

$r = x - b$; $d = 2x - y$;

$deltax = b - r$; $deltay = d - y$;

$theta = \tan^{-1}(deltax/deltay)$;

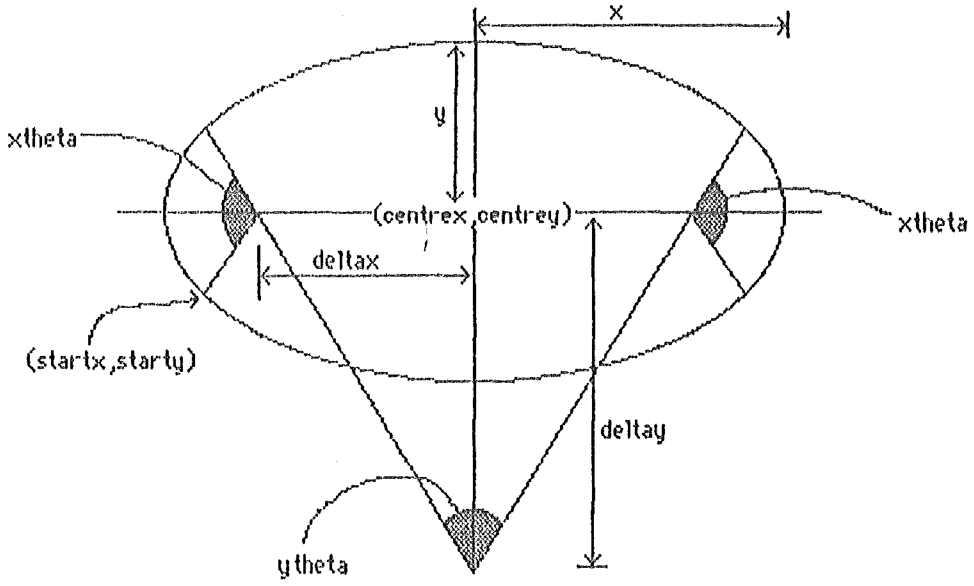
$xtheta = 2theta$;

$ytheta = 180 - xtheta$;

$startx = centrex - d*\sin(theta)$;

$starty - centrey - d*\cos(theta) + deltay$.

the appropriate values are shown below



As can be seen, the starting point, the arc centres, radii, and degrees can all be calculated.

Appendix C

Path Stretching Formula

Mathematics for Path Stretching

LAGS has start point (Sx,Sy) (typically the current cursor or object position) and the end point (Ex,Ey) to move to. It also has a coordinate sequence, which contains three or more coordinates, of which the start coordinate is (sx,sy) and the end point is (ex,ey).

To *stretch* the coordinate sequence so that (sx,sy) equals (Sx,Sy) and that (ex,ey) equals (Ex,Ey) the following calculations are performed.

```
scale_x = (Ex - Sx)/(ex - sx);
scale_y = (Ey - Sy)/(ey - sy);
if any of these get set to zero, then set from zero to 1.0;
offset_x = (Sx + Ex)/2 - scale_x*(sx + ex)/2;
offset_y = (Sy + Ey)/2 - scale_y*(sy + ey)/2;
```

Now to scale any point (x,y) in the coordinate sequence, all that must be done is to set x to

```
scale_x*x + offset_x
```

and set y to

```
scale_y*y + offset_y.
```

Appendix D

User Manual

User Manual

This User Manual is more correctly termed a reference manual as it does not explain all the ideas of LAGS that a typical user may require (for example the four types of database entities). It defines the syntax of all the commands recognised by LAGS, and explains the effect each has. Where appropriate, each command covered has a few syntactically correct examples, but no explanation as to what each example is given as they are considered self explanatory.

Explanation of the Syntax

All symbols and commands (or words) that LAGS recognises appear in **bold** form. Note that if a symbol is not bold, then it is not part of the command; for example ( ) may appear in both bold and non bold.
Any clause enclosed by [ ] is an optional part of the command.
Any clause enclosed by { } specifies that the clause may be repeated zero or more times.
Any clause enclosed by < > signifies the place where the parameters of the command are to be placed.
Any clauses separated by | signifies that there is an alternative; either one or the other.

All numbers must conform to the following syntax .....

```
    {+ | - } ( {<digit>} . <digit> {<digit>} ) |
             ( <digit> {<digit>} [ . {<digit>} ] )
```

thus the following are all valid numbers

```
             -123  45.67  .89  +.1  10.  -0.8
```

All names and identifiers must conform to the following syntax .....

```
             <alphabetic> { <alphanumeric> }
```

where currently a maximum of only 12 characters is allowed.

LAGS typically assumes that a command has been completed if all non-optional parts of a command has been entered & comes across something unexpected (say the next command). To force the termination of a command, the semi-colon character can be used.

All commands and identifiers are **case independent**.

To refer to a database entity, either the identifier of the entity can be used, or the entity number (which can be obtained via the index or display commands). For example if 'bert' is points identifier number 2, then
bert and **points** #2 are equivalent.

The clause "- **smooth | complete**" appears at the end of a number of commands. This is recognised by LAGS as correct syntax, but as far as the current implementation is concerned, this option is not implemented.

Arc Command

Syntax: **arc about** <x>,<y> **by** <degree> **deg**
or **arc through** <x1>,<y1> **to** <x2>,<y2>

This command draws an arc from the current position, either through point x1,y1 to x2,y2, or alternatively about points x,y by the given degree.

Examples: arc about 10,20 by 30deg;
arc through 10,20 to 40,50;

Circle Command

Syntax: **circle [at** <x>,<y>] **[with] radius [of]** <radius>

This command plots a circle with centre at x,y (or at the current cursor position if no coordinates are specified), with the radius specified.
The cursor is positioned at the centre of the circle after this command.

Examples: circle at 100,200 with radius of 50;
circle radius 10;

Clear Command

Syntax: **clear [screen]**

This command clears the screen.

Examples: clear;
clear screen;

Define Command

Syntax: **define points** <newpointsname> **as**
(<x>,<y>**)**[,] **(**<x>,<y>**)**[,] **(**<x>,<y>**)** { [,]**(**<x>,<y>**)** }
which defines a sequence of coordinates to be stored into LAGS for later use.
Note that at least three points must be specified.

Example: define points bert as (1,2),(3,4)(10,20);

or **define object** <newobjectname> **as**
{ graphic_commands } **quit**
which define a sequence of graphic commands as an object.
Valid graphic_commands in this instance are .....
arc, draw, move, plot, print, circle, and ellipse.
It should be noted that ordinary non graphic/animation commands are still executed within the definition of such an object. For example, commands such as help, index, where, set, system, and display are still recognised.
In the loading of an object in this way, the prompt from LAGS changes from '!' to '...' to signify that the user is still in 'define object mode'.

```
Example:    define object square as
                move to -1,-1;
                draw using (-1,-1),(1,-1),(1,1),(-1,1),(-1,-1);
                quit;
```

or **define function** <newfunctionname> **as**
a valid arithmetic expression involving 'x'
which defines a function as the corresponding arithmetic expression.
Previously declared functions may be used in these expressions, as well as the normal operators +,-,*,/ and // which is integer division.
Parentheses (,) are recognised. The precedence of operators, from highest to lowest is (,) ⇒ - (negation) ⇒ function ⇒ *,/,// ⇒ +,- .

```
Example:    define function cube as x*x*x;
            define function tricky as
                10//x + sin cos tan x - cos(x%10);
```

or **define format** <newformatname> **as**
a valid sequence of format specifications
defining a format specification used by the 'print' command, -the syntax has not yet been decided.

Display Command

Syntax: **display** <object>|<points>|<function>|<function> [**details**]

This command displays the definition of the specified entity.

Examples: display square;
display points #2 details;

Draw Command

Syntax: **draw to|by** <x>,<y> [**-straight**]
or **draw** [**to|by** <x>,<y>] **using** <points>|
(<function> [**from** <x1> [**to** <x2>] [[**in**] **steps** [**of**] <step>]])
[- **curved|straight**]

In the first case above, a straight line will be drawn from the current cursor position to the point specified, which can be relative to the current position, or absolute.

In the second syntax, a sequence of points is generated, according to the 'using' clause, and ...

In the case of '-straight' a series of lines is drawn through the points;

In the case of '-curved' a curve is fitted through these points.

The default here is '-straight' if points is used, and '-curved' if a function is used.

In this method, the starting and ending points of the function values/points are matched onto the current position and the ending position if to|by is used. Any points that lie in between these two extremes are "stretched" to conform to the shape required.

Examples: draw by 10,20;
draw to 700,200 using bert; where bert is a points entity
draw using sin from 0 to 20 in steps of 0.4 -straight;

Edit Command

Syntax: **edit** <points>|<object>|<function>|<format>

This command enables editing of the above 4 entities. It uses the editor on the system LAGS is implemented on. The information that is produced to enable editing is a syntax correct definition of the entity being edited. On completion of the edit, LAGS will read the new definition automatically.

Examples:
```
edit square;
edit function #10;
```

Ellipse Command

Syntax: **ellipse [at** <x>,<y>]
[with] radius [of] <x_radius> **and** <y_radius>

This command plots an ellipse with centre at x,y (or at the current cursor position if no coordinates are specified), with the x and y radii as specified.
The cursor is positioned at the centre of the ellipse after this command.

Examples:
```
ellipse with radius 60 and 40;
ellipse at 100,100 with radius of 20 and 80;
```

Equate Command

Syntax:
```
equate <newobjectname> as
     { <subobject> : <objectname> at <x>,<y>
     [(scaled [about <x>,<y>|centre]
          by <scale>|(<scale_x>,<scale_y>))
     [and rotated [about <x>,<y>|centre]
          by <degree> deg)]]
     }
quit
```

This command creates an object as an amalgamation of other objects. Each subobject of this object can then be accessed by the animation commands individually by 'dot notation', or the object can be treated as a whole. Obviously it is possible to have some of the subobjects defined as a group of other objects, and so on.
This is not yet implemented.

Fill Command

Syntax: **fill** [<object>] **with** <colour>

This command fills the specified object, (or the current object if none is specified), with the requested colour.
This is not yet implemented.

Example: fill square with black;

Index Command

Syntax: **index [all | objects | points | functions | formats]**

This command displays the index of the appropriate clause. If a clause is not specified, then the default of 'all' is assumed, which displays the index of all identifiers known to the system.

```
Examples:   index;
            index functions;
```

Load Command

Syntax: **load** "filename"

This command enables the loading or 'opening' of a file which can subsequently be executed via 'run'.

Example: load "testfile";

Move Command

Syntax: **move to|by** <x>,<y>

This command moves the graphics cursor from the current position to the point specified, either relative to the current point, or absolute.

Examples: move by 40,-5;
move to 100,34.6;

Place Command

Syntax: **place** <object> [**at** <x>,<y>] [**as** <newobjectname>]
[- **smooth|complete**]

This command places an object at x,y (or at the current cursor position if no coordinates are specified).
If the 'as' clause is present, a new object is created as copy of the one specified, and this is placed on the screen.
If '-smooth' is specified, the object appears on the screen as it is created, but if '-complete' is specified, the object will appear instantly; default is '-smooth'.
The cursor is then positioned at the centre of the object on the screen.

Examples: place square at 100,400;
place triangle at 30,40 as boris;

Plot Command

Syntax: **plot** [**at**|**by** <x>,<y>]

This command plots at the specified point x,y, which can be specified as relative to the current cursor position, or absolute.
'Plot' by itself causes a point to be plotted at the current position.

```
Examples:   plot;
            plot at 40,50;
            plot by -10,0;
```

Print Command

Syntax: **print** "<text>" [**at** <x>,<y>] [**using** <format>]
[**over** <num_frames> **frames**]

This command prints text at either the current cursor position, or the coordinates specified. If the 'format' clause is used, this specifies the attributes of the text displayed. If the 'over' clause is used, this causes the text to appear over that many frames -default is one frame.
The text itself can use an escape character '\', which is followed by a ' " ' it implies the character itself, and if followed by an 'n' a new line is implied.
The format entity is not yet implemented, but the rest of this command is.

```
Examples:   print "Hello World" at 30,50;
            print "Line 1\nLine 2";
```

Quit Command

Syntax: **quit**

If this command is entered while in 'object definition mode', it signals that the object definition is complete.
If, however, this command is entered in 'normal mode' then it signals the end of the current LAGS session. You are however asked if you are sure that you want to quit, replying 'no' will not terminate the session, but 'yes' will.

Remove Command

Syntax: **remove** <object>|<points>|<function>|<format> **[from database|screen]**

This command removes the appropriate entity from the database or screen. Obviously removing something from the screen is only valid for an object. If no 'from' clause is specified, the default of 'database' for points, function, & format, while 'screen' for an object is assumed.

```
Examples:   remove square from database;
            remove points *3;
            remove square from screen;
```

Rotate Command

Syntax: **rotate** [<object>] **[about centre**|<x>,<y>]
[(fixed|relative)]
by degree **deg [while** rotate_scaling_clause]
[over <num_frames> **frames] [-leaving [as [is]]]**

This command will rotate the object (if not specified, then the current object) by the specified degree.
The point that the object is rotated relative to is, by default, the centre of the object, but it can be specified by the 'about' clause. If this point is to remain relative to the object, presuming that the command moves the object relative to this point, then 'relative' must be specified -default is 'fixed'.
If the rotating is to occur gradually over a number of frames, then the 'over' clause must be used.
Use of the 'leaving as is' option causes all animation positions to remain on the screen, and thus leaves a trail where the object has been.
For details on the 'rotate_scaling_clause', refer to the transfer command.

```
Examples:   rotate square by 180deg;
            rotate square about centre (relative) by 80deg
                 over 30 frames -leaving;
            rotate square by 360deg while scaling by 1.05
                 over 10 frames;
```

Run Command

Syntax: **run** ["<filename>"]

This command will cause execution of the LAGS commands in the specified file. If no filename is specified, then the 'current known file', presumably specified by the 'load' command, will be used.

Examples: run;
run "testfile";

Save Command

Syntax: **save** "<filename>" [- **coded|ascii**]

This command is meant to save the current LAGS state for later reloading. The actual syntax as use has not been fully determined, so the command is not yet implemented.

Scale Command

Syntax: **scale** [<object>][**about centre**|<x>,<y>][**(fixed|relative)**]
by <scale>|(<scale_x>,<scale_y>)
[**while** rotate_scaling_clause]
[**over** <num_frames> **frames**] [**-leaving** [**as** [**is**]]]

This command will scale the object (if not specified, then the current object) by the specified scale factor(s).
The point that the object is scaled relative to is, by default, the centre of the object, but it can be specified by the 'about' clause. If this point is to remain relative to the object, presuming that the command moves the object relative to this point, then 'relative' must be specified -default is 'fixed'.
If the scaling is to occur gradually over a number of frames, then the 'over' clause must be used.
Use of the 'leaving as is' option causes all animation positions to remain on the screen, and thus leaves a trail where the object has been.
For details on the 'rotate_scaling_clause', refer to the transfer command.

Examples:
```
scale triangle by 2,4;
scale about 10,10 (fixed) by 10 over 40 frames;
scale about centre while rotating by 5deg
      over 6 frames - leaving;
scale square by 6,-1 while scaling about 10,20 by 3 and
      rotating about 200,300 (relative) by 6deg
      over 50 frames -leaving as is;
```

Set Command

Syntax: **set screen [to]** <x>,<y> **and** <x>,<y>

which sets the screen coordinate system according to the two coordinates specified - top left and bottom right coordinates.

Example: set screen to 0,1000 and 2000,0;

or **set dump off** | (**[on] [to** "<filename>|**screen]**)

which sets the dumping of information as commands are entered on or off. This is of more use to the "experienced" LAGS user to see if LAGS is doing what is expected.

Examples: set dump off;
set dump on to screen;
set dump to "tracefile";

or **set command trace**
off | (**[on] [to** "<filename>|**screen]**)

which sets the tracing of commands entered on or off.

Examples: set command trace off;
set command trace on to screen;
set command trace to "tracefile";

or **set graphics trace**
off | (**[on] [to** "<filename>|**screen] [only]**)

which sets the recording of all graphics codes produced by LAGS in response to the commands entered on or off. If on, this allows a file to be produced which can be later run at a faster speed. If the 'only' option is specified, then no graphics comes to the terminal at all - this would be useful in running a file through LAGS to produce just the graphics codes, for later viewing independent of LAGS.

Examples: set graphics trace off;
set graphics trace on to screen;
set graphics trace to "tracefile" only;

Store Command

Syntax: **store** <object>|<points>|<function>|<function> [**as** <newname>]

This command stores the entity in the database, and if the 'as' clause is used, stores it under that newname.
This command is implemented, but no database is yet.

Examples: store square as newsquare;
store square;

System Command

Syntax: **system** "<text>"

This command passes <text> to the operating system to execute the system command. Control is passed back to LAGS when the command has completed.

Example: system "ls"; Unix command to list file directory.

Transfer Command

Syntax: **transfer** [<object>] **to**|**by** <x>,<y> [**using** <points>|
(<function> [**from** <x1> [**to** <x2>] [[**in**] **steps** [**of**] <step>]])]
[**while** rotate_scaling_clause]
[**over** <num_frames> **frames**] [**-leaving** [**as** [**is**]]]

This command will transfer the object (if not specified, then the current object) to the specified coordinate, either relative to the current position, or absolute.
The path the object takes to move to its final destination from its current position is defined by the 'using' clause. If this clause is not present then a straight path is chosen.
Use of the 'leaving as is' option causes all animation positions to remain on the screen, and thus leaves a trail where the object has been.
If the movement is to occur gradually over a number of frames, then the 'over' clause must be used.

The rotate_scaling_clause is defined as .....

(rotating [about <x>,<y>|**centre] (fixed|relative) by** <degree> **deg)** |
(scaling [about <x>,<y>|**centre] (fixed|relative)**
by <scale>|(<scale_x>,<scale_y>))
{**and** rotate_scaling_clause}

The rotate and scaling values in this clause are in terms of each frame rather than in terms of being spread evenly over all frames.
The point that the object is scaled or rotated relative to is, by default, the centre of the object, but it can be specified by the 'about' clause. If this point is to remain fixed in relation to the object while it is being animated, then 'fixed' must be specified -default is 'relative'.

Examples:
```
transfer square by 100,200;
transfer square to 700,400 while rotating by 5deg
     over 30 frames -leaving as is;
transfer square to 500,600 using bert;
transfer to 500,600 using cos from -10 to 10 steps .5
     over 50 frames;
```

Transform Command

Syntax: **transform** <object1> **into** <object2>
[**over** <num> **frames**]

This command transforms one object into another, over a specified number of frames -default is one frame.
This command is not yet implemented.

Undo Command

Syntax: **undo**

This command undoes the previous command, if it can be undone.
Commands which can be undone by this command are ..... none at present.

Where Command

Syntax: **where**

This command displays the coordinates of the graphics cursor, along with the screen corner coordinates, as specified by the 'set screen' command.