THE UNIVERSITY OF CALGARY

MOGL: A MOTION GEOMETRY LANGUAGE

by

JOHN HARRISON

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF

COMPUTER SCIENCE

CALGARY, ALBERTA

OCTOBER, 1989

ISBN 0-315-61648-2

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "MOGL: A Motion Geometry Language" submitted by John Harrison in partial fulfillment of the requirements for the degree of Master of Science

K. D. Loose

Supervisor,
Dr. K. D. Loose
Department of Computer Science

[illegible]

Dr. A. W. Colijn
Department of Computer Science

[illegible]

Prof. T. P. Keenan
Department of Computer Science

[illegible]

Dr. W. B. Clark
Department of Curriculum and Instruction

October, 1989

Abstract

Motion geometry is currently taught to students using various hands-on and mathematical techniques which tend to obscure the underlying concepts. This situation can be improved by creating a computer environment in which to explore motion geometry. This thesis involves the research of various techniques including human-factors design, graphics, microworlds, and theories of human learning, to create this environment.

A prototype system, MOGL, is described. MOGL is an environment which allows the user to create and manipulate graphical objects using motion geometry functions (such as rotation, translation, enlargement, reduction, and reflection). New shapes can be constructed by combining previously defined shapes using programs written in the MOGL language. All this can be accessed through an interface incorporating a pointing-device, menus, and windows to display the results.

The design of MOGL is discussed, including why various techniques are used as well as how they are implemented.

MOGL is then evaluated according to user response, and how well the design goal of creating an effective microworld for the exploration of motion geometry is met.

Preliminary results of the investigation are encouraging, indicating that a complete version of MOGL should be implemented. Moreover, it is demonstrated that by allowing for the functioning of the user as part of the design of the system, by designing the system as a microworld, by paying a good deal of attention to the user interface, and by using more traditional computer science techniques, an effective computer subsystem can be created.

Acknowledgements

I would like to thank my supervisor Dr. K. D. Loose for his valuable advice and patience throughout this project. Thanks are also due to Professor T. P. Keenan for his encouragement and help, to Dr. D. B. Harrison for his proofreading, and to various students from the Shad Valley summer program and elsewhere for their help in the evaluation of the project. Special thanks to Professor R. R. Skemp for his insights into human learning, and for planting the seeds that eventually grew into this project.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1. The Current Situation

Currently, motion geometry (also known as transformational geometry) is taught by having students draw set shapes, and having them transform each point of the shape to a new position using formulae. This can be a tedious task, and often the underlying concepts of motion geometry can be lost in the tedium of formulae.

Some attempts are made to have students explore operations by using hands-on techniques to avoid dealing with formulae, such as using tracing paper or translucent mirrors (called MIRAs) to simulate the various operations. However, these techniques can be frustrating to the students, because of the difficulty of tracing an image. As well, many students find the operations pointless, doing apparently mindless activities (to satisfy the teacher).

Some motion geometry operations (such as the enlargement and reduction of shapes) do not translate well to hands-on activities. For these operations, students will again have to resort to formulae, plotting each point of the transformed shape individually.

1.2. Improving the Situation

Computers by their very nature are very useful for manipulating numbers; so the task of dealing in complicated formulae could be done by the computer, leaving the student with more of a chance to learn motion geometry concepts.

This task could be undertaken in many different ways. The computer could simply be used as a sophisticated calculator; with students entering the required formulae, and coordinates of each point, and the computer giving back the new co-ordinates so that the student could plot the points.

Alternatively, a computer environment to allow the transformation of a set shape could be created, showing the transformations using graphics and hiding the formulae from the student. Such an environment, called MOTIONS, has been created (Thompson, 86-92). The MOTIONS environment is a step in the right direction, allowing students to explore motion geometry concepts without too much additional "noise" (such as formulae or MIRAs), using both the mathematical and graphical capabilities of the computer. However, this environment is somewhat limited. Students still have to use a set shape (a pennant), and the transformations to enlarge and reduce shapes are not included. As well, the environment does not include a very good user interface.

This could be improved upon by creating a complete "microworld." Seymour Papert has formulated the concept of a microworld as a limited but consistent computer world for students to explore (Mindstorms, 117). This type of exploration allows students to expand their knowledge by experimenting with the system. Concepts could then be assimilated as described by Piaget (quoted by Papert; Mindstorms, 120). MOTIONS is a microworld by these definitions, but not necessarily as general as it could be.

A further step can be taken by integrating many different areas of computer science with this concept of a microworld to create a more useful environment for motion geometry than others currently available. This environment should allow all the major transformations, and it should allow students to create their own shapes. A good user interface must be created to allow effective interaction with the environment. This allows exploration of a more general version of motion geometry than in MOTIONS, with the experimental aspect of microworlds, and the ease-of-use of a good interface.

Perhaps an attempt could be made to model the concepts that the student using this system has, and to direct the student's activities according to "bugs" in the concepts that the student already has. However, as Brown and Burton have discovered (156), this can be a very complex and difficult task. This would be beyond the scope of this thesis.

So, the goal of this thesis will be to create an environment for the exploration of motion geometry, allowing all major transformations on shapes that can be arbitrarily defined. By using the idea of microworlds, combined with computer science and human factors design theories, this should become an effective computer subsystem.

This problem is a viable problem to study, as it will require investigation into many different areas of computer science. As well, it also will include some investigation of how the user of such an environment will operate, requiring research into learning and human-computer interaction. All of this can be brought together to create a useful computer subsystem.

1.3. Creating the Environment

Many pieces have to be put together to create this new environment.

First, the operations of motion geometry and computer systems conducive to learning will be investigated. Other computer systems using motion geometry systems will be explored to discover operations that are used, and to investigate techniques of presenting graphics. As well, the concept of microworlds will be investigated.

Second, human-computer interaction must be considered. When a new computer environment is created, the computer scientist must be aware of many aspects of the potential environment. The behavior of the environment must be planned. However, the potential interaction of the user with the environment must also be considered. Indeed, a view of the environment in use could be taken as having two systems interacting with each other. One is the computer system, and the other is the user (Hollnagel and Woods, 584). It is important to design a system that is actually useful to users, and which will help the users achieve their goals.

Third, to ensure that users of this system will actually be able to learn using the system, how students learn in general will be researched.

Once all of this research has been completed, it will be necessary to implement a prototype of the environment. To do this, the environment must first be designed. All of the major operations and the features the user has available will be defined at this point.

When the design is complete, it must be implemented on a computer.

To ensure that the prototype is actually useful, a survey of some users will then be done; having them use the program and determine how useful they found it in learning and using motion geometry concepts.

1.4. Scope of the Project

The environment to be created will be a prototype of a microworld for learning motion geometry. While the result will not be a complete system, it should be enough to give an effective idea of how a complete version would work.

By the careful design, implementation, and testing of this environment, it will be discovered whether following the theories chosen will actually produce an effective environment. If successful, this will allow the future creation of a complete sub-system for learning motion geometry. More importantly, success would guide future researchers along a design path shown to work in at least one area. If unsuccessful, then it will be known that this design path can lead to a dead end.

The success of the project can be judged by the reaction of users of the system, and by comparing the completed system to the original design goals. Problems related to the system being a prototype can be factored out, and then an overall assessment of the success of the project made.

The goal of this thesis, then, is to create a computer sub-system according to certain design principles from computer science and certain theories of learning. This system should allow users to effectively explore and learn about the world of motion geometry.

1.5. Organization of the Thesis

The current chapter details the need for this research. Chapter two investigates motion geometry, computer science, and human-computer interaction concepts (including some learning theory). Chapter three integrates these concepts into a general design for the environment. Chapter four is a description of the detailed design. Chapter five is a description of the implementation of the system. Chapter six is an analysis of the system, including the testing of the system and suggestions for improvements. Finally, chapter seven investigates whether or not the the goal of the study has been reached.

Chapter 2

Background Literature

2.1. Introduction

The goal of this thesis is to produce a prototype computer subsystem in which users can explore and learn motion geometry concepts. A review of work from several areas pertinent to this project will be presented under three topics.

First, graphics systems based on motion geometry will be presented. Second, human factors design will be reviewed with respect to good techniques for presenting information to the user. Finally, some related research on theories of how people actually learn will be presented.

2.2. Graphics Systems

As the prototype system being designed is to be used for motion geometry, graphics will be necessary. This section presents three representative systems used for dealing with graphics. GROPER is a motion geometry based graphics system. Logo and MOTIONS are microworlds designed to investigate geometrical concepts.

2.2.1. GROPER

GROPER is a motion geometry based graphics language, developed by Brian Wyvill. Wyvill defines a graphics language as "A language which defines and manipulates pictorial data in a way comparable to that performed by the operation of a procedural language on numerical data" (Wyvill, 2). Here, the concept of a special-purpose language designed only for graphics is introduced, with GROPER being designed as a language of this kind.

2.2.1.1. General Description of GROPER

GROPER (GRaphics OPERating system) allows the user to create and manipulate graphical primitives called "pictures."

A picture is a graphical object made up of other pictures, with the most basic picture being the pre-defined picture *line*. Each picture must be defined by referring to other pictures. To make up a picture called "square," for example, the *line* picture can be called up and added to the picture as many times as needed. When adding a *line* to the square, it may be transformed so that the line is placed in the appropriate position. Once the square is defined, it may be used to create other pictures.

2.2.1.2. Transformations in GROPER

GROPER allows many different kinds of transformations on shapes, including translation (using the command 'ORIGIN'), magnification (using 'MAGNIFY'), rotation (using 'ROTATE'), and a general transformation function called 'BETWEEN' (Wyvill, 110-112). These allow linear transformations of a shape so that it can be positioned as necessary. The 'BETWEEN' function is especially interesting, as it allows a transformation to be defined as that which would move a single line from an arbitrary position on the screen to a different position, and then apply that transformation to the whole picture.

One common motion geometry transformation that GROPER does not include is reflection. This additional operation would be useful in a system to explore motion geometry concepts.

GROPER allows recursive definitions of shapes. That is, a shape can be made up of smaller versions of itself. As part of this feature, a recursion limit can be be set. Thus, a shape could be defined as being made up of copies of itself, without worrying about infinite recursion. The recursion limit is set to an integer value, and the recursion will only go to the specified depth.

2.2.1.3. Implementation of GROPER

GROPER is implemented as an interpreter. Version 3.2 runs on a Unix system, and uses an English-like command language (Greenberg and Wyvill). For example, to turn a line 90 degrees and add it to a picture of a square, the command would be:

```
.add line to square rotate by 90
```

GROPER is highly interactive. A shape that is being created can be viewed at any time, and errors can be detected as the user makes them. It is a good environment for an artist with a basic understanding of co-ordinate geometry, as lines can be placed where required in a picture using the basic motion geometry commands described above. The concept of multiple recursion also allows very complex pictures to be created, such as shapes which are made up of smaller copies of themselves.

2.2.1.4. Summary of GROPER

GROPER is an effective system for artists to create pictures using motion geometry. However, it is a system to help in drawing pictures. It is not designed as a way to learn motion geometry, and knowledge of some geometrical concepts is necessary before the system can be used.

Two main features from GROPER seem especially meaningful in designing other systems: the idea of using various motion geometry commands to position lines in pictures, and the notion of putting several commands together to create a program which allows the creation of a picture.

Geometry Toolbox (Allen, 3-4) also uses these two major features in constructing three dimensional manipulations.

2.2.2. Microworlds

Seymour Papert defines a microworld as "a little world; a little slice of reality" ("Microworlds," 79). A microworld is a model of some portion of the real world, but without all the noise that would be present if that concept was dealt with in real life. The concept of a microworld is well demonstrated in the programming language Logo, designed by Papert, and discussed in his landmark book, Mindstorms.

2.2.2.1. What is a Microworld?

Papert describes a microworld as a limited environment that is laden with possibilities for discoveries. It is non-threatening. In a microworld, there is no right and wrong, only what "works" or "doesn't work" in producing a desired effect.

A student can be left with a microworld to explore it, and to discover what is embedded in it without the need for programmed instruction or textbooks. Students using microworlds are in control of their own learning processes.

An advantage of microworlds over other forms of computer-based instruction (such as drill-and-practice and video games) is that they include the "factors that compel" as discussed by Lewin (274). That is, the student can set a goal, perhaps use attractive graphics in the system, use it in many different ways (bringing imagination into the activity), get feedback, and find the system rewarding in itself (i.e. not requiring an external motivation).

2.2.2.2. Building Concepts Using Microworlds

Fragmented knowledge can be very useful in a microworld environment. A student may discover a new piece of information about a microworld (say a new command in Logo), and then use that new fragment of knowledge as a basis for experimentation to discover more. By discovering more about the microworld, the student will discover more about the concepts embedded in the microworld. Students can start with very little understanding about a concept, and then build it up into much fuller understanding by using their own initiative.

A student can also have a more intimate connection with knowledge gained from a microworld. A student can become very attached to a concept, and use that as a motivation to find out more about it.

2.2.2.3. Logo

An important feature of a microworld, and one particularly relevant to the field of motion geometry, is that objects between solid objects (in actuality) and abstract objects (in math and physics) are available.

Logo uses the idea of turtle graphics to allow students to explore a geometrical microworld. This turtle is a link between the idea of a real turtle, and the abstract ideas of geometry. The turtle appears as an arrow on a computer screen, and can be instructed to move forwards and turn. The turtle leaves behind a "trail" showing where it has moved. By commanding the turtle to move in different directions, students can build up geometrical pictures. This type of geometry is called "turtle geometry."

2.2.2.4. MOTIONS

Patrick W. Thompson (86-92) describes a microworld called MOTIONS, designed for the teaching of motion geometry. It was designed so students could understand motion geometry as a mathematical system. It allows transformations of a pennant in a Cartesian coordinate system. Commands are available to translate, rotate, or flip (reflect) the pennant.

Transformations can be composed in several ways in MOTIONS. First, several commands can be placed on a single line, allowing the user to watch each individual command occur in rapid succession. Second, the commands can be enclosed in brackets, so that only the final result of applying all the commands is visible. Third, a new command can be defined, incorporating several commands.

Students can use MOTIONS with a set of problems designed to take them through various transformations and understand the concepts of motion geometry.

2.2.2.5. Summary of Microworlds

Papert envisions that at some point in the future, there may be complex networks of microworlds, containing many sectors of knowledge. Papert believes (perhaps somewhat optimistically) that this could replace the current concept of "curriculum."

It should be possible to create a microworld based on motion geometry. A system could be created which allows exploration of various motion geometry concepts. It may motivate students to learn new concepts, for the avenues of exploration that a new concept can open up. And the objects on the computer screen may be good for illustrating abstract mathematical concepts in a way more real than dealing with numbers and points on paper.

2.3. Human-Computer Interaction

Of the research available on the interaction between systems and humans, three areas seem particularly important. These are user modelling by the system, the user's perception of the system and the design of interfaces.

2.3.1. User Modelling

Hollnagel and Woods (583) state that a Man-Machine System (MMS) must be designed by taking into account the cognitive functioning as well as the physical functioning of the user. A system designed to provide a motion geometry environment would deal primarily with the cognitive functioning of the user, so this issue is a very important one.

The computer system must somehow facilitate the exploration of motion geometry concepts by a constantly changing user (Brown and Burton; Paliès et. al.; Park and Tennyson).

Ideally, the system should build an explicit model of how the user functions (e.g. a representation of the user's concepts), and use this model to determine exactly what the user would need to explore. A different system would be presented to each user, depending on the individual's needs (Rich, 200). The system could even change as the user understands more. This technique has been used in systems where the interaction of the user with the system is relatively easy to describe (Greenberg and Witten, 31). However, initial investigations into this area showed that developing an explicit model of a user's understanding of even a "simple" task (such as addition) can be difficult (Burton, 182). Modelling a user's motion geometry concepts would be beyond the scope of this thesis.

It would be comparatively easy to design a system for the canonical user (i.e. a system for a "typical" user of the motion geometry environment), rather than trying to model each individual user. A degree of stereotyping within this model could allow a distinction between novice and expert users.

It would be desirable to understand the mental functioning of a "typical" student, so that an environment appropriate for exploration could be provided. To attempt to address this issue, section 2.4 contains an investigation of two theories of human learning.

2.3.2. The User's Model of the System

The user will form a model of any computer system that is used, forming mental models of how the system works (Carroll and Thomas, 107). It is important to ensure that the user forms appropriate models. Carroll and Thomas give eight recommendations, including the following:

Metaphors, or analogies to explain the operation of the system, should be explicitly stated to novice users of a system (Carroll and Thomas, 111-113). The metaphors should cover as many aspects of the system as possible. The limits of the metaphors should be made known to the user. The possible consequences to the user and the system should be taken into account. Thus, when the motion geometry system is being designed, it is appropriate to develop analogies to explain how the system operates to the user. For example, a metaphor for enlargement could be developed by having the user image that the shape is printed on a balloon that is being inflated. The screen could be presented as a window onto a part of this balloon. Then, if the shape disappears all together, the metaphor helps the user understand that the shape has been enlarged so much that it is outside of the confines of the window.

By keeping these design principles in mind when designing the system and the supporting documentation, the system should be easier for a novice user to use.

2.3.3. Interface Design

The various design principles for a human-computer system have been described by Hill (6-12). Thirteen principles are cited which will guide the development of a good user interface.

The principles applicable to this thesis are as follows:

1. Know the user

2. Design the tools the user needs

The first two principles are of primary importance. Firstly, the designer should become acquainted with exactly what the user needs. This is similar to building a canonical user model into the system, as described in section 2.3.1. Secondly, the tools designed should be those the user needs, suited to the tasks at hand.

3. Make the system easy to learn and remember

Short term memory is limited to about 7±2 items (Miller, 14-43), so this must be catered to in any presentation of multiple items. As well, the system should be designed according to the models that a human will most naturally use.

4. Avoid both human and machine failure modes.

Deal with errors in a positive way to avoid machine or user "breakdown."

5. Structure the interaction

The interface should be structured in a way easy for the user to follow, so that commands can be easily found in menus.

6. Convey a feeling of control to the user

Provide feedback on what the system is doing.

7. Consider division of labour
8. Take into account human performance limits
9. Provide facilities to monitor system activity

By following these principles, an interface could be designed which allows effective interaction between the student and the computer.

2.4. Theories of Learning

As this thesis involves the design of an environment to promote learning, two pertinent theories of learning have been chosen as paradigms to follow. These are relevant parts of the theories by Jean Piaget and R. R. Skemp.

2.4.1. Jean Piaget

Piaget's theory (Piaget and Inhelder) shows a way of looking at the intellectual development of humans that emphasizes the mental processing humans do. Application of these theories by R. R. Skemp and others in educational environments have produced some very encouraging results, as discussed in section 2.4.2. Some of these concepts seem particularly applicable to the design of a computer system which will better meet the needs of those using it.

2.4.1.1. Schemata

As a human develops, the intellect is organized into schemata, or structures representing the outside world. This organization allows the human to adapt to the environment. This adaptation is a two-way process.

The first part of adaptation is called assimilation. Here, all the information that comes into the human through the senses is modified so that it can be made to fit existing mental structures.

The second part of adaptation is called accommodation. This is the process of making new connections in existing schemata as a result of the information from the assimilation process. Connections observed in the outside world allow connections to be made internally.

2.4.1.2. Phases of Development

Piaget hypothesized a sequence of development which appears to occur for every human. The phases of development may not occur exactly at the ages described, but they always seem to come in the same sequence. It is important to be aware of the stages when designing educational systems. Only those stages relevant to this thesis will be discussed.

The stages when a child would be most likely to learn most motion geometry concepts are the concrete operations and formal operations stages.

The concrete operations phase occurs from 7 to 11 years of age. In this phase, the child has not yet learned formal reasoning, but can reason with operations relating directly to concrete objects.

As well, a child is also able to classify objects in this phase, creating concepts by combining similar examples. The child is able to form a semi-lattice, or a hierarchy, of concepts, similar in structure to a tree. Each element in this hierarchy is known as a class.

The child's mental operations at this stage exhibit the characteristics of a group, as follows:

1) closure - combining any two or more elements of a set produces another element of the set.

2) associativity - the order elements are combined does not affect the result.

3) general identity - one element of the set, when combined with any other element, leaves the result unchanged.

4) reversibility - for every element, there is another element, called its inverse, which when combined with the original element, produces the identity.

Piaget calls these structures groupings, because in addition to the mathematical properties of groups, they exhibit two additional special identities, as follows:

1) resorption - Each class may have sub-classes underneath it in the hierarchy. If any of these sub-classes is added to the original class, the result will be the original class.

2) tautology - Each class is an identity for itself, because adding a class to itself results in the original class.

In the concrete operations phase of reasoning, a child relies on direct experience to solve problems. This can lead to incorrect reasoning. A higher form of reasoning is needed to be able to analyse the operations that have been built up, seeing ways in which other analyses may be possible rather than only accepting what has been experienced. This is the formal operations phase of reasoning, usually occurring after 11 years of age.

The thinking a child does in the formal operations phase is also more coherent, bringing the two logics of classes and relations into a single logical way of thinking.

2.4.1.3. Summary of Piaget

A motion geometry system can benefit from the work of Piaget. Since this system would be a learning environment, it should allow opportunities for a student to assimilate the concepts being presented, and to accommodate them into schemas being produced.

The phases of development are also important. This system is likely to be used in students learning in the Concrete Operations phase (if it is used in an Elementary school), and in the Formal Operations phase for older students.

Older students and adults should find it easier if the geometrical concepts are first presented in a concrete way, and then presented allowing formal operations to be carried out. So, the system should allow a progression from the concrete to the formal.

In this way, the motion geometry concepts could be learned in a natural way.

2.4.2. R. R. Skemp

Professor R. R. Skemp has synthesized a comprehensive theory about the process of learning, with origins traceable to the work of Piaget. This theory has been used with much success in school environments. Activities based on this theory have been created by Skemp (Structured), and these have been shown to help children learn more effectively (Marilyn Harrison and Bruce Harrison, 34; Bruce Harrison et. al., 297).

This section is a summary of those parts of the theory applicable to creating an environment for exploring motion geometry.

2.4.2.1. Goal Directedness

Skemp's theory (Intelligence; Psychology) recognizes that much of what humans do is goal directed. That is, humans will try to achieve desirable states, or goals, and avoid undesirable states, or anti-goals. This is similar to the idea of goal-directedness in computer systems designed to mimic human intelligence, such as STRIPS and ABSTRIPS (Cohen and Feigenbaum, 523-530) and SAM and PAM (Barr and Feigenbaum, 306-315).

Goals can exist in a hierarchy. For example, a student may want to draw a picture on a computer's screen. To attain this goal, the student will have to set several sub-goals, such as positioning lines, or learning how to draw certain shapes.

If the student is operating under goals that he or she has set, rather than being forced to do activities, the student is more likely to learn.

To achieve the goals that are set in this motion geometry system, the student will build up a set of concepts related to motion geometry.

2.4.2.2. Reality Testing

It is important to be able to test the accuracy of one's concepts. This process is called "reality testing."

There are three forms of reality testing.

In the first and most basic mode of reality testing, one tests against his or her expectations of events in actuality. In the second mode one tests against the realities of other people (e.g. by discussion). The third mode involves testing for consistency with other knowledge and beliefs that one already has.

In the design of a motion geometry system, allowances should be made for all three modes of reality testing.

2.4.2.3. Noise

Noise refers to everything that is not relevant to the activity being learned. This factor can influence how easily a concept is learned. The less noise that there is, and the more that an activity is contributing directly to the formation of a concept, the better the concept will be learned.

Skemp (Intelligence) maintains that when a concept is first being formed, a minimum amount of noise should be present. When a concept is better formed, more noise can be tolerated. And finally, as a concept's formation is continuing, a higher noise level is optimal so that the student is better able to separate the concept from the noise.

2.4.2.4. Summary of Skemp

A motion geometry system should be created which allows goals to be set (such as the goal of drawing a picture), and which allows a student to construct and test an accurate representation of the actuality of motion geometry in his or her reality, using all three modes of reality testing.

The system should also promote learning, allowing students to experiment to learn new concepts and applications. Noise should be kept to a minimum as much as possible in the early stages of learning, but can be increased (with concepts such as programming) later on in the learning process.

2.5. Summary of the Literature

A system to encourage the learning of motion geometry can be created, building on computer science concepts illustrated in GROPER and in Human-Computer Interaction research. However, some attention must also be paid to research done in the field of education and cognitive psychology to create a system which will be effective for the people who will be using it.

By putting all of these concepts together, it should be possible to create a very effective motion geometry system.

Chapter 3

Design Issues

3.1. Introduction

The goal of this thesis is to produce a computer subsystem based on the concept of a microworld. This would allow users to explore motion geometry concepts. Additionally, the system will be used for a wide range of users. At one end, it will be used for children from the ages of 10 through 14. This corresponds approximately to the transition from the concrete operations phase to the formal operations phase as described by Piaget. Thus, the students using this language will be quite comfortable with classifying concepts, and will be getting ready to deal with more complex operations than those exhibited by a grouping. An effective microworld for Motion Geometry will allow for the development and use of these operations. This system will also be used for older children and adults, who find the formal operations natural.

The microworld produced in this thesis will be an interactive computer language called MOGL, for a MOtion Geometry Language.

3.2. MOGL's Roots in Logo

Papert's microworld, Logo, has been found to be very useful in introducing children to basic concepts in mathematics and geometry, and is used widely in the public school system. Commands in MOGL will have the same format as commands in Logo (Abelson and Klotz, 25-41; Abelson, 175-193), but rather than using Turtle geometry, MOGL will include commands based on motion geometry. By basing the design on Logo, students who already know Logo will only have to learn motion geometry commands to use MOGL. As well, LOGO has been extensively used and modified, and is therefore a good format to build upon.

While the control flow and variables in MOGL will be much the same as in Logo, the graphics commands will be radically different. The main focus of the discussion will be on the criteria for making design decisions.

3.3. The Basic Concepts Used in MOGL

The MOGL language will allow the exploration of motion geometry concepts. Thus, the design of the language will allow a person to use basic motion geometry concepts to build up a picture on the screen, similar to MOTIONS and GROPER. In this way, the user can set a goal of trying to draw a specific picture, and then learn and use the motion geometry commands to help achieve this goal.

The language is designed to be easy to use for a student in the concrete operations phase of development. So, objects to be manipulated will be available on the screen.

A picture will be drawn by invoking a shape, and moving this shape to the appropriate location on the screen; perhaps changing its size and orientation. This shape will be called a "sprite," thus building an explicit model for the user of a dynamic shape.

Once a sprite is moved to an appropriate location on the screen, it will be possible to give a command which will leave a "footprint" behind. In this way, when the sprite moves on, a mark in the shape of the sprite will be left behind on the screen. Again, an explicit model of what happens is available to the user.

By moving the sprite around, and leaving footprints behind it will be possible to build up pictures of arbitrary complexity.

MOGL will be written as an interpreter, as it would often be used in this "direct mode" (just issuing single commands and watching the results).

The above would provide a basic system for students in the concrete operations phase of development.

For higher level students using formal operations, the system will include the capability to create new sprites. This can be naturally introduced into the language by allowing programs to be written. Thus, the user will be able to give, as a program, the moves needed to make a new sprite. These moves will be given as combinations of commands and footprints. Once the program is completed, the name of the program will be typed, and a new sprite would appear based on the footprints given in the program (a mode is available where the program is simply stored and can be called up later).

This will produce a real challenge, and many interesting possibilities for a more advanced student to explore. Motion geometry may be learned on a deeper level, through more complex operations and greater noise. As well, computing concepts are easily illustrated, such as order of execution of commands, looping, and recursion.

This progression of learning could be used just as well for an adult as for a child. First, experimentation could progress in the direct mode, corresponding to concrete operations. Second, the adult could learn to write programs, corresponding to formal operations.

3.4. The motion geometry operations to be used in MOGL

Since MOGL is designed to provide an environment in which to learn geometry, the basic motion geometry operations taught in the public school curricula will be available. These are similar to the operations that GROPER uses, with the addition of reflection. The main operations to include would then be:

1) Rotation - rotating the sprite around some point.

2) Translation - "sliding" the sprite to a new position on the screen.

3) Enlargement / Reduction - changing the size of the sprite.

4) Reflection - reflecting the sprite through a flat "mirror."

These operations have traditionally been taught by using paper and pencil and moving shapes around by translating each point individually. The translations are carried out by applying mathematical formulae to each co-ordinate. The concepts being illustrated tend to get lost in the noise from having to apply the formulae. To avoid this, MOGL will use simple commands that execute in a obvious way.

The commands will be simple in form (such as *turn*, *slide*, *enlarge*, *reduce*, and *reflect*) with arguments to indicate the degree of transformation.

Each of these commands will have the basic group properties (i.e. closure, associativity, general identity, and reversibility), for easy use by students in the concrete operations phase of development.

For the rotation command, *turn*, one argument will be necessary; that is, how far to turn. This will be given in degrees, since it is the most common measure of angles in the schools.

The translation command, *slide*, will require two arguments. These could be given in rectangular or polar coordinates. The decision was arbitrarily made to use polar coordinates. The arguments will be given as the distance first, then the direction.

The commands to change the sprite's size, *enlarge* and *reduce*, will each require a single argument. In this case, the identity will be one, since it makes sense to think of making a shape multiples of itself. Thus, an enlargement of two would result in a shape twice as large.

Finally, the reflection command, *reflect*, will require some sort of "mirror" to be available to reflect the shape in. This mirror will appear as a dashed line on the screen, and could be moved to anywhere on the screen. It is possible that some sort of activity will require multiple reflections in several mirrors, so several mirrors will be made available. Thus, the reflect command will require one argument; that is, the number of the mirror to reflect the shape in.

3.5. Sprites in MOGL

Since MOGL allows the user to create sprites using programs, only the basic building blocks will be required as the elementary sprites. A library of programs allowing the user to use any number of more complex sprites could be programmed using MOGL.

The most basic building block in graphics is the pixel. So, a sprite could be made available consisting of a single point. However, this sprite would have limited applications, since building up a shape out of single points would be quite tedious, and it is likely that the transformations on the object would appear quite slow (as every single point in the resulting shape would have to be transformed individually to a new location).

The next step up from a pixel is a simple line. This makes more sense to use as a basic building block, as most shapes that elementary students would like to create are likely to be made up of straight lines. If a point is ever required, the line sprite could be reduced until it is a line of negligible length.

An advantage of using a line as the basic building block is that in any linear geometrical transformation, only the end points of the line have to be transformed. This saves transforming every pixel in a picture to get a new picture.

The basic sprite in MOGL, then, will be a line. This is similar to the basic "picture" in GROPER. Based on this line, many shapes could be created.

The size of this line as it initially appears would be one unit, and it would appear with one end at the centre of the screen.

3.6. Types of Values Available in MOGL

As MOGL is a programming language, various values will be required in the writing of programs. Here, the design is based on Logo, where five different types of objects are available:

1) Booleans - logical true or false values.
2) Integers - positive and negative whole numbers; written as a sequence of digits possibly preceded by a negative sign (for example, -43).
3) Reals - decimal fractions; written as an optional negative sign followed by a sequence of digits, with a decimal point (for example, 3.1415).
4) Words - a sequence of characters; starting with a double quote, and with no spaces or punctuation (for example, "fred).
5) Lists - a sequence of words (without double quotes), separated by spaces. These are contained within square brackets (for example [fred joe jim]).

Note that lists and words are not the same thing; a word is an element of a list (a list consisting of one word is not a word, it is a list containing a word).

A variable name is given as a word; so if the user is referring to the name of a variable, it would be preceded by a double quote. However, the value of a variable is referred to as a colon followed by the alphanumeric part of the variable's name. This keeps references by name and by value distinct. An example of where this might be used is when the name of a variable rather than its value is to be passed to a procedure.

In Logo, variables are not typed; and don't have to be declared before they are used. This same convention is followed in MOGL. However, some operations (such as multiplication) will require a certain type of argument. Thus, a run-time error will occur if a variable does not have the correct type of value at the time the operation is executed.

3.7. Shape Design and Recursion

The goal of producing a complex shape can be broken up into several sub-goals of making simpler shapes, and then putting all of the shapes together to produce the final object. This naturally falls into developing several small modules, and integrating them as a large program (a bottom-up approach). A shape could even be made up of smaller copies of itself. This is a natural way to build up complex geometrical shapes, and can also prove to be a very easy introduction to recursion.

Students will concentrate on breaking a problem down into sub-problems; and solving those sub-problems with separate programs. Thus, it is easy for a student to create a hierarchy of goals (as in Skemp's theory), and explicitly break down a goal into several sub-goals. Similar operations could be done using iteration, so this will also be available in MOGL.

3.8. The User Interface

For a good environment, a good user interface is needed. This interface must allow exploration in transformational geometry with a minimum of unnecessary noise.

It would be counter-productive to force the user to type in commands to do each operation. This presents extra noise, in requiring keyboarding skills. Most beginning users (especially children) may find it easiest to use some sort of pointing device such as a mouse or a light pen as an input device. However, such options are limited due to the hardware available, and are not implemented in this project. Instead, step keys will be used to move a crosshair on the screen to simulate a pointing device (see section 3.8.3.).

As well, if the commands were to be typed in, this implies memorizing the commands. This is also extra noise, which can be eliminated by allowing the commands to be selected from menus. More advanced users, however, may wish to enter the commands on the keyboard for speed. Thus, the user interface will be designed with this in mind; to require a minimum of typing for the novice, but allowing typing to be used by the expert.

The system will be set up allowing selections to be made from a window (for novices), but allowing experts to type their commands in without needing to use a pointing device. This is effectively designing a canonical model of the user, but allowing the user to choose whether to use novice or expert interaction at any given point.

3.8.1. The Format of the Screen

The user interface for the system will attempt to present all pertinent information from MOGL in an easy to understand format. This structuring of the information will follow the 13 principles of human-computer interface design (Hill, 6-12).

The screen will be divided into four windows, with different functions. These are:

1) Graphics window
2) Menu window
3) Text window
4) Message window

The first window is to contain the graphics. That is, it will allow the user to see the shape being used. Since this is the main function of MOGL, this window will be in the centre of the screen.

The second window is used to present a menu to the user. This will appear to the right of the graphics screen. Since a light-pen or touch screen may be used for input at some stage, this is the best location for a right-handed user to make a selection without covering the graphics area with his or her hand. This window should be moveable to the left side of the graphics screen for left-handed users (allowing a degree of user modelling).

The third window, to the left of the graphics screen, is used to show the text of the commands being entered. So, as a user selects commands from the menu or types in commands, they will appear in this window. Therefore, even as a user makes selections from the menu, he or she can see how the text of the commands looks if entered from the keyboard.

The fourth window, at the bottom of the screen, is to show messages from the system. This would be used to indicate any errors, and to give any other messages from the system (such as whether programs were loaded into MOGL successfully or not). Of course, error messages will be positive and instructive, as described by Hill (9).

These four windows will allow the user to work in direct mode with MOGL using "user friendly" techniques.

3.8.2. Menus

The menu system will allow the user to access the majority of the graphics commands, so that shapes can be called up, manipulated with all the possible transformations, and made to leave footprints behind without having to resort to the keyboard. This would make the system easy to learn for the novice, and caters to students who may not be able to type effectively.

The system will also follow the general rule that humans can only deal with 7±2 concepts in short-term memory at a time (Miller, 15-43) so each menu should have a maximum of nine selections available.

Thus, all the commands won't be available on the top level. Instead, like commands will be grouped together, and a sub-menu used to access the commands.

On the main menu, there will be a selection to indicate that a transformation of the shape is desired. A sub-menu will then appear with all the possible transformation commands, from which one could be selected.

Similarly, a sub-menu will be used to select a sprite. The names of all of the sprites currently available will be available on this menu (it may require several screens to show all the sprites if many are defined).

3.8.3. Pointing Device Input

The easiest way to use MOGL through the user interface would be to make selections from the various menus by pointing to them. This requires some kind of pointing device to point to the selections desired.

One kind of pointing device is a touch screen. The user could point with his or her finger at the operation desired. Alternatively, a light pen could be used to point to a specific operation. If either of these pointing devices were used, it would be good to have the monitor mounted inside a desk so that the user has a horizontal surface to work with. This reduces fatigue in using a light pen or touch screen.

This sort of technology is not available at many schools, so MOGL must be designed for the possibility of using a less expensive form of positional input.

One such device would be a mouse. Compared to other inexpensive devices such as text keys, step keys and a rate-controlled joystick, the mouse is the fastest and most accurate pointing device (Card et. al., 1978). The disadvantage of a mouse is that it can take quite a bit of practice to discover how a pointer moves on a screen as a device is moved on a desk. Some people find this separation confusing.

Due to hardware limitations, step keys will be used to simulate input from a mouse in this prototype version of MOGL.

3.9. Summary of the Design Issues

The design of MOGL will be based somewhat on LOGO, but allowing motion geometry operations instead of the turtle geometry used in LOGO. All the essential motion geometry operations will be available. Sprites can be combined to produce new sprites, using programs consisting of geometrical transformations. Various other objects and techniques are included to simplify programming using MOGL. The language is designed to be interactive, so that commands can be executed as they are typed. Many commands can be put together to allow more complex geometrical operations and the definition of new sprites in programs. As well MOGL can handle recursive programs.

All the geometrical operations in MOGL are designed to "make sense," so that the user can understand what is happening during the transformation of a sprite. Algorithms will be chosen to maximize execution speed during manipulations of the sprite.

Finally, a good user interface will be provided between MOGL and the user, to allow easy interaction and learning. This interface will include windows for different types of output from the program, menus of commands, and pointing device input.

Chapter 4

The MOGL Language

4.1. General Analysis of the MOGL Language

A typical session in MOGL might include calling up a line sprite and transforming it in various ways. Each possible transformation is taken from the concepts of motion geometry. When the user has moved the line to the appropriate place, a mark can be left behind. In this way, pictures can be built up. The user can also define his or her own sprites by writing simple programs.

This chapter presents the structure of MOGL and defines the parameters and actions for each command. Any parameters for commands will be given within parentheses, to distinguish them from the actual commands.

4.2. Objects in MOGL

There are several different types of objects in MOGL that the user can use. The most important is the sprite, which can be transformed in various ways. However, the attributes of this object such as its orientation and location are not available to the user, the sprite is simply visible on the screen.

Other kinds of objects will be required if programs are to be written in MOGL. These are objects to represent words and numbers, which can be assigned to variables and used in various ways.

4.2.1. Assignable Objects Available in MOGL

Numbers in MOGL are either reals or integers. An integer is simply a string of digits, and a real is a number with a decimal point (there must be at least one digit before the decimal point). These numbers can be used anywhere that a number makes sense in the MOGL language (i.e. as values for variables, and arguments for certain commands).

It is also possible to use true/false values in MOGL.

MOGL also allows the use of words. A word is a sequence of alphanumeric digits, prefixed by a double quote. These can be used as variable names, for example.

Finally, it is possible to use lists in MOGL. A list is any sequence of numbers and characters enclosed in square brackets. These can be used to store strings, or sequences of commands.

4.2.2. Types in MOGL

MOGL does not require the user to explicitly define the type of a given variable before assigning a value to it. All type checking is done as the commands are being interpreted, and any inconsistencies (such as trying to add an integer to a word) are reported at run-time. This is in keeping with the way that Logo is usually implemented.

4.3. Commands Available in MOGL

The commands in MOGL can be broken into several logical groups. These are:

1) Basic graphics commands which allow the user to call up a sprite, and make transformations.
2) Control commands which allow looping and branching. These can be used directly or in programs.
3) Programming commands which allow the user to write named programs.
4) File commands for storing and recalling programs.

5) Miscellaneous commands that implement housekeeping tasks.

4.3.1. Basic Graphics Commands

These commands are the heart of the system. They are the commands that help the user understand the principles of motion geometry.

The *line* command simply calls up a simple sprite consisting of a straight white line, one unit long. This line appears with one end at the centre of the screen, and the other end one unit to the right.

The *enlarge* and *reduce* commands have the syntax:

```
enlarge {amount}
reduce {amount}
```

(here, {amount} is a numerical expression, as defined in section 4.5.1.). The *enlarge* command makes the sprite {amount} times as large (e.g. *enlarge 4* would make the sprite four times as large). The sprite is enlarged around the centre of the screen, NOT around the centre of the sprite. This decision was made because it would be difficult to calculate the centre of an arbitrary sprite, slowing down processing time more than necessary.

The *reduce* command is the inverse of the enlarge command. It causes the sprite to shrink around the centre of the screen. If a shape is enlarged, then reduced by the same amount, the original shape will be restored.

The *turn* command has the syntax:

```
turn {degrees}
```

It allows the sprite to be rotated around the centre of the screen. The expression gives the number of degrees to be rotated. The sprite will be rotated in a clockwise direction; if a counter-clockwise rotation is desired, the argument simply needs to be negated.

The *slide* command is used to move the sprite to a different location on the screen. It has the syntax:

```
slide {distance} {direction}
```

The sprite will be moved {distance} units in the direction given by {direction} degrees, where 0 degrees is to the right, and 90 degrees is upwards.

The *reflect* requires a mirrorline to be visible before it is used. There are eight mirrorlines available, arbitrarily numbered 0 through 7.

To activate a mirrorline, the command:

```
mirror {number}
```

is used. This calls up mirror {number}. This mirrorline will appear along the x-axis, and can be moved with the commands:

```
mturn {number} {degrees}
mslide {number} {distance} {direction}
```

These commands work exactly like the *turn* and *slide* commands, except that the first expression gives the number of the mirror to be transformed. In this way, the user is using concepts previously learned with sprites to move the mirrors.

The *reflect* command is used to reflect the sprite through an arbitrary mirrorline. It has the syntax:

```
reflect {number}
```

where {number} gives the number of the mirrorline to be used.

A mirrorline can be moved to any position on the screen, and then the sprite can be reflected in a given mirror. Because up to 8 mirrors can be active at once, complex sequences of reflections can be done.

If the user decides that a mirror is not necessary, the command:

```
hidemirror {number}
```

can be used. This command will remove the given mirror from the screen.

If the user wishes to use a transformation not included in the language, the *matrix* command can be used to allow any kind of linear transformation. This command has the syntax:

matrix {exp1} {exp2} {exp3} {exp4} {exp5} {exp6}

where {exp1} through {exp6} are numerical expressions.

Each point on the sprite, given by (x, y), is taken as a 1x3 matrix, p:

$$p = \begin{bmatrix} x & y & 1 \end{bmatrix}$$

This matrix is then multiplied by a 3x3 matrix, M:

$$M = \begin{bmatrix} \{exp1\} & \{exp2\} & 0 \\ \{exp3\} & \{exp4\} & 0 \\ \{exp5\} & \{exp6\} & 1 \end{bmatrix}$$

and the first two elements of the resulting 1x3 matrix give the new x and y co-ordinates of the point.

Once the user has transformed the sprite to the desired position, the sprite can be made to leave a "footprint" behind. This is done using the *mark* command. This command has the syntax:

mark

and will leave a grey line on the screen in the current position of the sprite. This line is fixed, and will not move under further transformations of the sprite.

Thus, pictures of arbitrary complexity can be built up by transforming the line sprite to various positions, and leaving marks behind.

4.3.2. The Control Commands

The control structures available in MOGL are similar to those available in Logo, and allow repetition and decision making.

Often, when writing a program, a user will want to repeat some task many times. The *repeat* command provides a simple way of doing this.

The *repeat* command has the syntax:

repeat {exp} [{commands}]

Here, {exp} is a numerical expression, and {commands} is a list of commands that are to be executed, enclosed in square brackets.

The expression is evaluated, giving the number of times that the action is to be repeated. Then the list of commands is executed that number of times.

A user may wish to execute some command conditionally on the result of some comparison. The *if* command allows this, and can be in one of the following two forms:

if {bool-exp} [{commands}]

if {bool-exp} [{commands}] [{commands}]

Here, {bool-exp} represents a boolean expression (as defined in section 4.5.2.), and {commands} represents a list of commands.

In the first form of the *if* command, the conditional expression is evaluated. If it has the value TRUE, then the list of statements is executed. Otherwise, the list of statements is ignored.

In the second form of the *if* command, the conditional expression is also evaluated. If it is true, then the first set of statements is executed. However, if it is false, the second set of statements is executed.

As the statements to be executed must be contained within square brackets, the problem of matching a set of statements to an *if* command is avoided.

MOGL also includes some commands for saving and retrieving programs.

At any time, all the programs that have been defined can be saved in a file. This is done by simply issuing the *save* command. It has the syntax:

```
save "{filename}
```

where {filename} is the name of the file in which the programs are to be saved. All of the programs defined so far will then be saved in whatever directory was active when the user started MOGL, under the name {filename}.mogl, overwriting any file by that name.

As well, extra programs can be added to the MOGL environment by using the *load* command, in the form:

```
load "{filename}
```

Here, {filename} is a word which gives the name of the file that the programs are to be loaded from. If the file {filename}.mogl exists, it will be opened, and all the programs it contains will be read in. If a program in the file has the same name as a program that already exists, it will be skipped over, and a message printed.

4.3.4. Miscellaneous Commands

Several additional commands are available for various other tasks.

The *print* command allows the user to print the values of variables, or constants, and to print out text messages. It can be used in one of two forms:

```
print {expression}
print [{text_string}]
```

In the first form, the expression {expression} is evaluated, and the result is printed. In the second form, all of the text within the list is printed, but the brackets are not printed.

This second form allows the user to output text messages to the text window.

Another command, the *clearscreen* command, is used to clear the graphics screen. It has the syntax:

```
clearscreen
```

and it erases everything on the graphics screen, with the exception of the active sprite, and any active mirrorlines.

The *view* and *noview* commands allow the user to specify whether or not to show each sprite as it is created by a program. A complex sprite may take a long time to be drawn on the screen if several smaller sprites have to be created and transformed. If this is a concern, the user could issue the *noview* command, and only the finished sprite will appear when a program is run. The default state can be restored with the *view* command.

However, it may be tedious to see a complex sprite being created. So, the *noview* command can be given so that the new sprite will appear instantly rather than being created line by line on the screen.

4.4. Variables

A variable in MOGL can contain a number (real or integer), a list (any text inside square brackets), or a word (a single word preceded by a double quote). Variables are not typed; so any of these types could be assigned to any variable.

Variables are set using the *make* command, as follows:

```
make "{identifier} {exp}
make "{identifier} "{word}
make "{identifier} [{text_string}]
```

Here, {identifier} is a character string representing the variable's name.

In the first format, the numerical or logical expression {exp} is evaluated, and the result is assigned to the variable.

In the second and third formats, a single word {word} and a list {text_string} are assigned to variables, respectively.

The value of the variable can be extracted by using:

:{variable_name}

Wherever this occurs in a MOGL program, the value of the variable is taken. So, if a number variable is part of an expression, the number contained in the variable is used in the expression. An interesting consequence of this is that a list variable could be used as the second parameter for a repeat command if the variable contains a list of the commands to be repeated. The language functions this way for completeness, so that all objects can be used in any appropriate place.

Since the variables are not typed, the variables do not have to be defined before they are set, and if a variable that contains one type of data has another type of data assigned to it, there is no problem. However, a variable must have a value before it is used. Otherwise, an error message will be printed. A discussion of the scope of variables is contained in the programming section (4.6.).

4.5. Expressions

MOGL allows the standard kinds of mathematical operations to be performed on numbers and variables. Expressions can return either a numerical or a boolean result.

4.5.1. Numerical Expressions

A numerical expression in MOGL can be made up of constants and variables. These can be combined using the operators "+" for addition, "-" for subtraction, "*" for multiplication, and "/" for division. Expressions can also contain parentheses.

MOGL uses the algebraic order of operations when evaluating expressions. First, anything inside brackets is evaluated. Next, any multiplications and divisions are done (from left to right). Then, addition and subtraction is done.

An expression can be used anywhere a number is allowed in a MOGL program.

The type of the result of a mathematical operation depends on the type of the arguments. If both arguments to the addition, subtraction, or multiplication operations are integers, then the result will also be an integer. If either or both of the arguments are reals, the result will be a real. The result of division is always a real.

4.5.2. Boolean Expressions

The results of two numerical expressions can be combined to form a boolean expression. This will result in a TRUE or FALSE value being generated. This is generally used for the *if* statement, to determine if a sequence of operations should be undertaken.

Numerical expressions can be combined using the operators "<" for less than, ">" for greater than, and "=" for equality. As well, these operators can be combined to get "<=" (or "=<") for less than or equal to, ">=" (or "=>") for greater than or equal to, and "<>" (or "><") for unequal to.

4.5.3. Constants

Integer constants are simply made up of a sequence of digits, without any spaces or punctuation between them.

Any number with a decimal point in it (even if no digits follow the decimal point) will be considered to be a real.

4.6. Programming

There are two modes of operation within MOGL, direct and programming mode. So far, only the direct mode of MOGL has been discussed. In this mode a prompt, ">," is given, and any of the commands explained above can be entered directly into the interpreter. The commands will execute immediately.

However, a sequence of these commands can also be entered for future execution and stored as a MOGL program.

A program in MOGL is like a procedure in other programming languages. Any number of MOGL programs can be available at once, and they can call each other arbitrarily. The words "program" and "procedure" are used interchangeably in MOGL.

4.6.1. Simple Programs

A simple program can be created by entering the command:

```
to "{program_name} {"{variable_name}}*
```

where {program_name} is a word representing the name of the program being created, and {"{variable_name}}* represent the names of zero or more variables that the program will use for parameters.

When this command is entered, the prompt will change to "?", which indicates that a program is being entered. Now, the program can be typed in as a sequence of direct commands.

To exit this mode, the user must type the command

```
end
```

on a line of its own to indicate the end of the program.

Now, whenever the name of the program is typed (with expressions for the parameters, if required), the sequence of commands in the program will be executed. The program is essentially a new command.

4.6.2. Programs to Create Sprites

Programs can also be written which produce new sprites. This is one of the most interesting features of MOGL, since it allows the user to define his or her own graphical objects to use. Rather than leaving static footprints behind (as in the regular programs), each *mark* command defines a part of a new sprite.

These programs work in the same way as a regular program, except that the program is first defined using the *sprite* command rather than the *to* command:

```
sprite "{program_name} {"{variable_name}}*
```

Here, a program is produced with the given name, and it is understood to represent a sprite. This kind of program is entered in the same way as a regular program.

When a sprite program is executing, however, a new sprite is being defined. Each *mark* command in the sprite program leaves behind a footprint which becomes a part of the sprite being defined. Thus, any kind of sprite can be defined by the user.

A user could build up a library of sprite programs, which could be used in the creation of many different kinds of pictures. The following program could be used to define a square, for example:

```
sprite square
    line                    (call up the line sprite)
    slide 0.5 90            (move it off centre)
    slide 0.5 180
    repeat 4 [              (repeat four times...)
```

```
                    turn  90                (turn the sprite 90 degrees)

                    mark                    (leave a mark behind)

               ]

          end
```

Of course, this new sprite could be used within another program which defines sprites; so that sprites of arbitrary complexity could be created. Note that the current implementation of MOGL does not allow comments as part of programs; the comments above were inserted into the program listing for convenience in explanation.

4.6.3. Recursion

Any program name could be used as a command within another program, or within its own body. As a result of this, recursion is possible. For example, a program could be written to compute the factorial of a number, recursively; putting the result into a global:

```
          to factorial "n

               if :n = 0 [                   (if n = 0)

                    make "f  1               (factorial is 1)

               ] [                           (if n<>0)

                    factorial (:n - 1)       (compute factorial of n-1)

                    make "f  :f  *  :n       (multiply this by n)

               ]

          end
```

Then, if the program is called as:

```
          factorial  4
```

the global variable "f will be set to 24.

4.7. Variables in MOGL

There are three major classes of variables in MOGL:

1) Global variables

2) Local variables

3) Parameters

4.7.1. Global Variables

By default, any variable used is a global variable. Global variables never need to be declared in MOGL; any variable name can be used in a MAKE statement, and that variable will be created (if necessary) and have a value assigned to it.

The command:

```
make "x 4
```

would, if x is not defined yet, make a global variable called x, and would assign the value 4 to it. This variable would then be available in the direct mode, as well as in all procedures.

4.7.2. Local Variables

A global variable can be used anywhere in the program, no matter where it is defined. This could cause a lot of problems with side-effects from two procedures using the same global variable for different purposes. To solve this problem, local variables are also available in MOGL.

A local variable has to be declared within the procedure that it is local to, using the *local* command with the name of the variable as a parameter. Thus the command:

```
local "x
```

will set up a local variable in the procedure in which the command is used.

This variable is available in the procedure defining it, and in any procedures that this procedure calls.

As soon as the procedure with a local variable is finished, however, the variable ceases to exist.

4.7.3. Parameters

Finally, parameters are also used in MOGL. These are defined at the start of a procedure, and give storage for the values passed into the procedure. These parameters have the same scope as if they were defined as local variables within the procedure.

Parameters only allow values to be passed into a procedure. Thus, if the value of a parameter is changed within a procedure, no effect is made on the calling program's variables. Thus if the following procedure were defined:

```
to demoparams "x
        print :x
        make "x 4
        print :x
end
```

and the procedure called as follows:

```
make "y 5
demoparams :y
print :y
```

then the following output would be obtained:

```
5       (printing the value of x inside the procedure; as passed from y)
4       (printing the new value of x within the procedure; y should not be
        changed)
5       (printing the value of y after the procedure is completed; it has not
        changed)
```

In this way, the parameters have no side-effects.

However, sometimes it is necessary for a procedure to change a global variable, for example if the procedure is being used as a function of some sort. Using the definition of scope that has been used, it is possible to do this without having to use the global variable's name directly within the procedure. This could be done by passing the **name** of the variable to the procedure rather than its value.

So, the above program could be rewritten so that that value of y is changed. Then, the name of the variable to be changed (y) can be passed to the procedure as a parameter:

```
to demoparams "x
    print :x          (x contains the name of the variable to be
                      changed to 4)
    make :x 4         (make the variable whose name is in x equal
                      to 4)
    print :x
end
```

and the procedure called as follows:

```
make "y 5
demoparams "y
print :y
```

The output would be as follows:

```
"y      (this is the value of x; it contains the name of variable y)
"y      (the value of x hasn't changed)
4       (however, y is changed to 4)
```

So, using these techniques, the names or the values of variables can be passed into procedures.

Care must be used when passing names of variables in recursive procedures, as what the name represents may change as the procedure is returning.

4.8. The User Interface

The user interface to MOGL is designed to make MOGL into an environment that can be used by both novices and experts to explore the motion geometry microworld.

When MOGL is started up, the user is presented a screen with four separate windows (fig. 1).

The TEXT window is used for the user to type commands. The user can use any of the MOGL commands just by typing on the keyboard, and the command will appear in the TEXT window, and execute.

Any drawing that is done in MOGL will appear in the GRAPHICS window. If shapes are larger than the window, only the parts within the window will be visible. There is no facility to zoom out to see objects outside of the window.

The MESSAGE window, on the bottom, will be for any status or error messages coming from MOGL.

Finally, the window labelled MAIN is the menu window, and it is currently showing the main menu. A crosshair will be available on the screen, and arrow keys can be used to move this crosshair to any position on the screen. When the crosshairs are positioned on the appropriate menu entry, the spacebar can be pressed to make the selection from the menu. Any commands that are selected will appear in the TEXT window (as if they had been typed), and will execute.

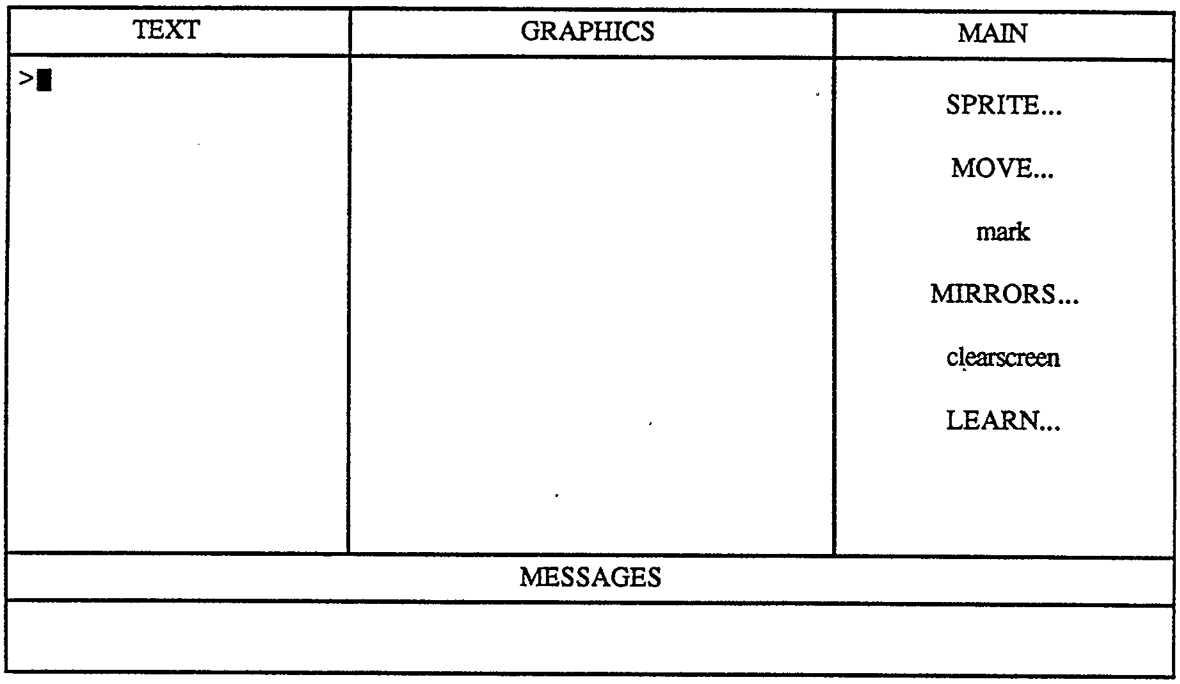


Fig. 1 Windows in MOGL

If a selection is made from the menu, the word selected will flash, and the action will take place. Any of the words in lower-case letters are names of commands. Words in upper-case letters are new menus that will appear when selected. If a word in the menu ends in three dots, this means more information will have to be given before the command is complete.

So, if the user selects *clearscreen* from the menu, the graphics screen would clear immediately.

If the user chooses *SPRITE...* from the main menu, a new menu will appear, listing all of the available sprites (fig. 2).

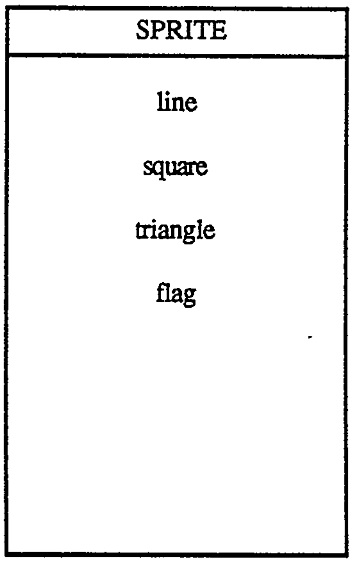


Fig. 2 The Sprite Menu

The user could then select the desired sprite, and it would appear on the screen. This list of sprites is updated as the user defines more sprites.

If the user selects *MOVE...* from the main menu, the MOVE menu will appear (fig. 3).

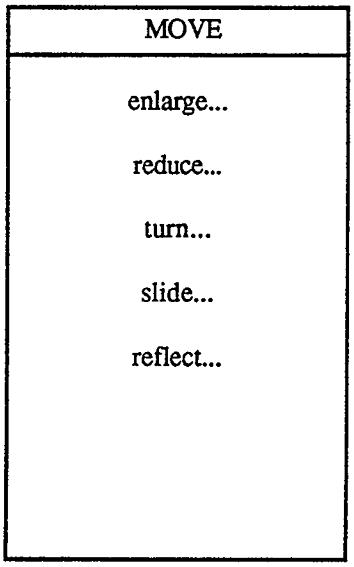


Fig. 3 The Move Menu

This menu lists all of the available motion geometry commands. Each command requires at least one parameter, and that is supplied by going into further menus. For all of the parameters that are *not* angles, the menu presented will consist of a scale. So, if the user selects *enlarge...*, the enlarge window will appear in the menu area, with a scale (fig. 4).

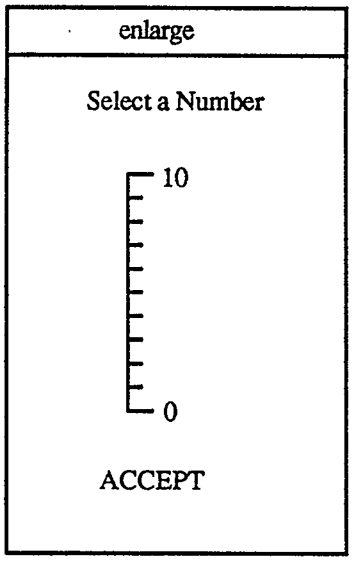


Fig. 4. The Enlarge Window

Here, the user can select any number from the scale by moving the crosshair to the number and pressing the space bar (an arrow will appear with the selected number beside it). When the desired number is selected, the user can select *ACCEPT* to complete the operation. Only integers can be selected from the scales.

If the user, however, selects *turn...* from the MOVE menu, a dial will appear. This dial allows the user to select an angle to be turned (fig. 5).

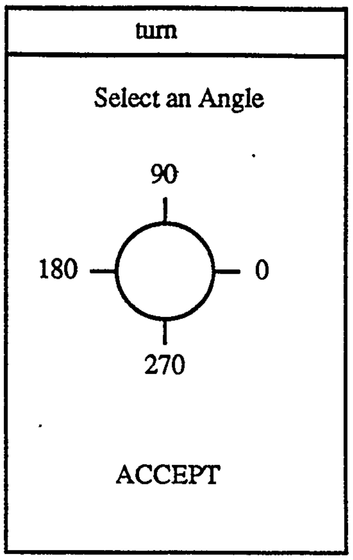


Fig. 5 The Turn Window

Here, the user can point to any point on the circle, and a line will appear at that point with a number indicating the selected angle. Once the desired angle is selected, the user can select *ACCEPT* to complete the operation. The available angles are in 15 degree increments.

Each of the commands on the MOVE menu use either the dial or the scale (as appropriate), or a combination of the two. The slide command, for example, uses a scale (to indicate the distance to slide) followed by a dial (to indicate the heading for the slide).

Another menu that is available from the MAIN menu is the menu allowing the definition of mirrors. If the user selects *MIRRORS...* from the MAIN menu, the mirror menu will be available (fig. 6).

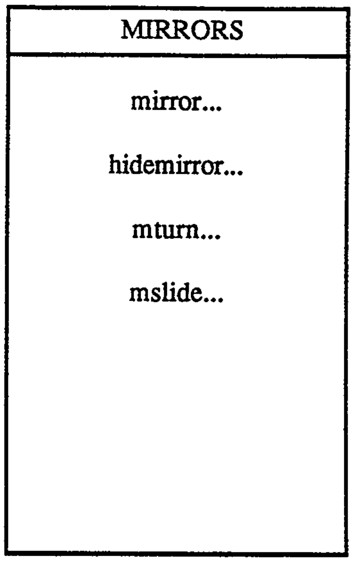


Fig. 6 The Mirror Menu

Here, the user can select one of the mirrors to make it visible (using *mirror...*), hide one of the mirrors (using *hidemirror...*), or position a mirror (using *mturn...* and *mslide...*). These commands use scales and dials to enter values, as appropriate.

Finally, if the user wishes to create a sprite using the user interface, he or she can select *LEARN...*, and a whole new screen will overlap the old one, to allow the commands for the sprite to be entered (fig. 7). This facility is currently available for the *sprite* command, but not for the *to* command.

The LEARN screen is just like the original screen, with the exception that the clearscreen command isn't available (that shouldn't be necessary when defining a sprite), and an *end...* command is available to end the program.

The user can now enter the moves to make the sprite on this screen, and be able to see how each command works as it is being entered. This is a vast improvement on conventional programming in MOGL by typing commands without any graphics appearing on the screen. Even users who are comfortable with typing all their commands could select this screen to be able to create their program while watching how the sprite looks (typed commands will still work, and will become part of the program).

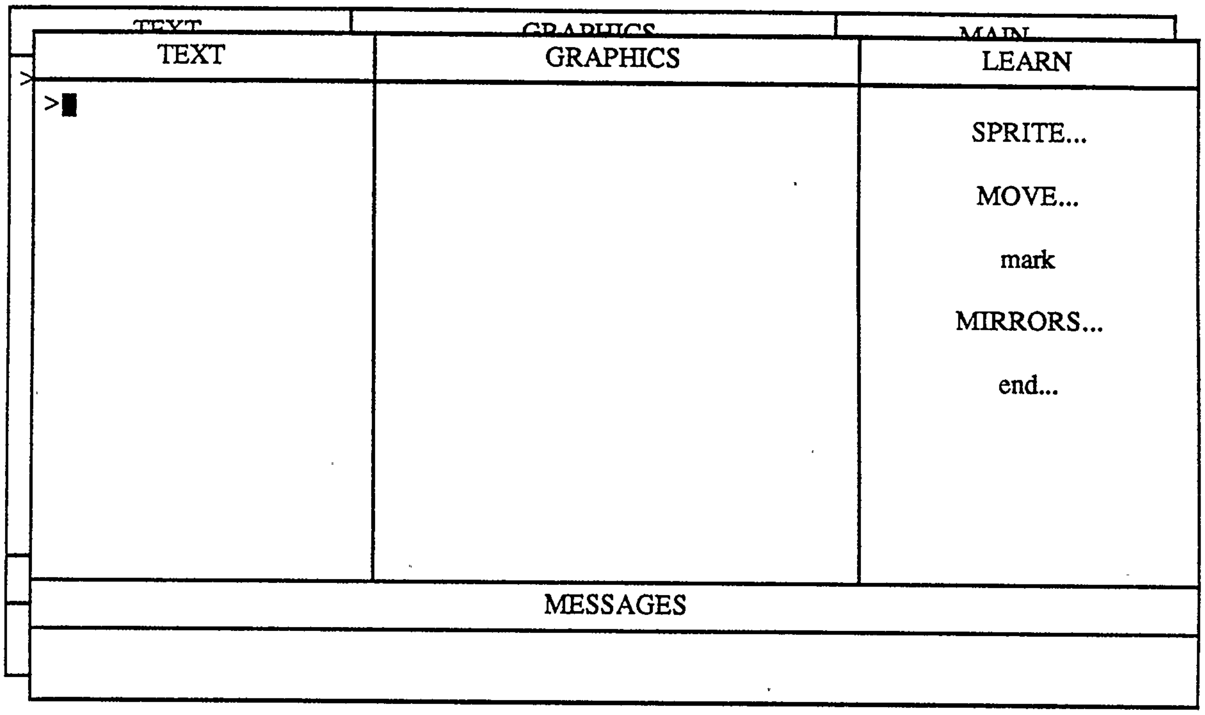


Fig. 7 The LEARN Screen

Once the user has defined the sprite, he or she can select *end...* and the original screen will appear. The user will be asked for a name for the sprite in the message window, and the new sprite will be added to the sprite menu.

Notice that this feature is just for creating simple sprites with no parameters. Recursive programs and parameters will not be allowed in this mode.

With this menu structure, the novice user can explore much of the power of MOGL without typing a single command. The expert user may also use the interface for the structured output, and the LEARN feature. All normal MOGL commands can be typed in the main mode, and programs may be typed in without using the LEARN mode.

Chapter 5

Details of Implementation

5.1. Choice of Implementation Language

If MOGL is to be used in the schools, it should be written so that it can eventually be moved to the computers that the schools use. As a result of this, using a fourth-generation language such as LISP to implement it in would be counter-productive as LISP requires a support environment that would tax the limits of micro computers that the schools use.

So, it was decided to write MOGL in C, as C is flexible and compilers exist for many machines. This should mean that the language is quite portable; requiring mainly modification of the graphics portion (and this can be minimized by careful planning).

A preliminary version of the language has previously been implemented in C on an Apple][computer by the author (Novel), and a rudimentary user interface implemented on a VAX computer (User). From these two experiments, MOGL has been significantly re-structured, and a complete prototype of MOGL with the full user interface has now been implemented on a Sun computer, designed to use a Visual 550 terminal for output.

5.2. Structure of the Interpreter

This section is a discussion of the various programming considerations when developing the MOGL interpreter.

5.2.1. Mathematical Basis for Motion Geometry

The variety of geometry used in MOGL is a linear transformational geometry.

Thus, for any of the operations on lines of a shape, it is only necessary to transform each of the end-points of the line, and draw a line through the new end-points. General curves are not allowed in MOGL, but could be simulated creating shapes consisting of many short lines.

Each of the points is given as an (*x*,*y*) co-ordinate; which defines the location of each point in the conceptual plane used by the program. This location is then translated to appropriate co-ordinates for the output device being used.

The transformation of each point can be accomplished through the use of simple mathematical formulae. For example, to enlarge the shape, the number for each *x* and *y* co-ordinate can be multiplied by a constant.

Rather than using a unique formula for each transformation, it is possible to use a general matrix multiplication; using different matrices to get different transformations (Foley and Van Dam).

Each (x,y) co-ordinate can be represented by the matrix:

$$p = \begin{bmatrix} x & y & 1 \end{bmatrix}$$

This matrix, *p*, can be multiplied by a 3x3 matrix, *M*, to achieve a new matrix, *p'*, as follows:

$$p' = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

where *x'* and *y'* are the points after transformation.

In other words:

$$p' = p \, . \, M$$

Various matrices can be used for *M*, to achieve different transformations, as follows:

1) Rotation

To rotate a shape around the centre of the screen through an angle, *a*, the following matrix is used:

$$M = \begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2) Translation

To translate a shape *n* units in the direction *a*, the following matrix can be used:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ n*\cos(a) & n*\sin(a) & 1 \end{bmatrix}$$

3) Enlargement

To make a sprite *n* times as large, the following matrix is used:

$$M = \begin{bmatrix} n & 0 & 0 \\ 0 & n & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The reduction matrix is the same, except that to do a reduction of *m* units, *n* can be defined as follows:

$$n = 1 / m$$

4) Reflection

To reflect the point in a line running through (x,y) that is rotated by a degrees, define:

$$s = 1 - 2 * \sin(a) * \sin(a)$$

$$c = 2 * \sin(a) * \cos(a)$$

Then, the matrix to use is given by:

$$M = \begin{bmatrix} s & c & 0 \\ c & -s & 0 \\ x*2 & y*2 & 1 \end{bmatrix}$$

A procedure called "exmatrix()" takes as arguments the numbers in the first two columns of M (as the final column is always the same). This procedure transforms the current sprite. So, to write a procedure to execute a motion geometry command, it is only necessary to read in the arguments from the motion geometry command, and call the exmatrix() procedure with the appropriate arguments, as described above. This simplified the writing of the geometry portion of MOGL considerably.

5.2.2. Input to the Interpreter

Input to the interpreter can come from direct commands typed on the keyboard or selected from menus (supplied a line at a time by the user interface), or from stored programs.

5.2.2.1. Direct Input

When operating in direct mode, the user interface prompts the user for a line of input, and once the user has typed the line it is passed to the interpreter.

The interpreter has a procedure called getsymb() which scans the line a character at a time, scanning for symbols. This procedure returns the type of symbol found; returning a lexeme if it is a command (indicating the appropriate command), or a lexeme indicating what kind of object it is (e.g. LIST, WORD, REAL, etc.). Expressions are evaluated, and the results are stored in a global for future use.

The symbol returned by getsymb() is interpreted. Thus, if it is a command, a C procedure is called which executes the actions necessary to carry out the command (possibly calling getsymb() again to get the values for any arguments). If it is the name of a program, the appropriate program is executed.

These actions occur for every line the user types; completely interpreting the previous line before the next line is typed. A side-effect of this approach is that each line must have complete commands; one of the arguments to a command can't occur on the line entered after the command in direct mode (in program mode, commands can be split across several lines). Several commands could be given on the same line, however, separated by spaces.

5.2.2.2. Interpreting User Programs

If the user enters a command that indicates that a program is to be entered, the prompt changes to show that the user is entering a program. This program is stored as a text string, which can be interpreted later. This string is stored in a data structure which also includes the name of the program (for when the interpreter is looking up a particular program), and pointers to the locations of the main body of the program, and the names of the parameters the program uses.

When a program is to be executed, the parameters to the program are evaluated and stored as variables (any variables with the same names as the parameters are saved as described in section 5.3.2.). The old text input position, and pointers to any shapes that were being created are all stacked. This old text input position may be NULL (indicating direct input from the terminal), or it may have indicated that text was being read from another program.

Now, the text from the new program is used as the source of input, and the commands are executed as if they were typed in direct mode. If the program is defining a sprite, the position of any lines *marked* are added to a data structure.

When the *end* command is reached, indicating the end of the program, the sprite pointers, text input position, and variables are all popped off the stack. If the program was defining a sprite, the current sprite is set to the sprite defined by any *marks* left by the program.

5.2.3. Error Processing

All of the C procedures used to interpret commands (except those dealing with interpreting expressions, as described in section 5.4.5.) return a value to indicate if an error has occurred while interpreting the command. If an error is encountered at any point, the stack is unwound, and control returns immediately to the top level of the interpreter. An error message is printed indicating the type of error that occurred, and all the structures containing programs, variables, and so forth are left intact. No debugger is available.

5.3. Data Structures

Four main data structures are used within the MOGL interpreter. The functions of these are:

1) to store a sprite being defined or being used
2) to store variables
3) to store programs
4) to provide a stack.

5.3.1. Storage of Sprites

A sprite is stored as a linked list, with each element in the list containing a pointer to a structure defining a single line in the sprite (line), and a pointer to the next line in the sprite (next).

A pointer (shape) is kept to the currently active sprite, and to any sprites being defined by a program.

Each line in the sprite is stored as four pointers, which point to double accuracy real numbers indicating the x and y co-ordinates of the end-points of the line in the conceptual plane (x1, y1, x2, and y2). If the MOGL program is moved to a microcomputer, the most accurate numbers available should be used to avoid rounding error accumulating during transformations of the sprite.

The structure for a sprite consisting of two lines is diagrammed in figure 8.

5.3.2. Storage of Variables

Variables are stored in a hash table with chaining. A doubly-linked list is used for the chain to make deletions simple. The variable structure for two variables hashing to two different numbers is shown in figure 9.

Here Table is the hash table. Each entry in table is a pointer to a chain of variables. Each variable in the chain has a name (represented as a string), a type (either integer, real, or string; represented by a number), and a value pointer. The value of the variable is pointed to by the appropriate pointer, according to the type of variable. If it is an integer variable, an integer pointer (ivalue) points to the value. Similarly for real (rvalue) and string (svalue) variables.

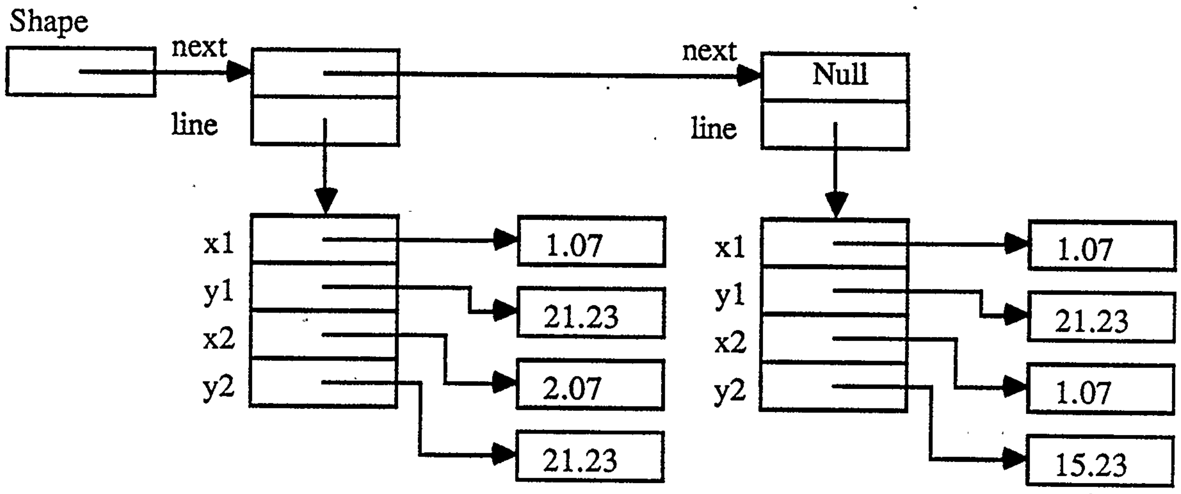


Fig. 8 Data Structure for a Two Line Sprite

Each variable also has a pointer to the next variable in the chain, and the previous variable in the chain.

The data structure represented in figure 9 contains two variables; an integer variable called "count" containing the number 42, and a list variable called "alist" containing the list, [demo].

Along with this data structure are procedures to add a variable to the list; to delete a variable from the current list; and to find the current value of a variable.

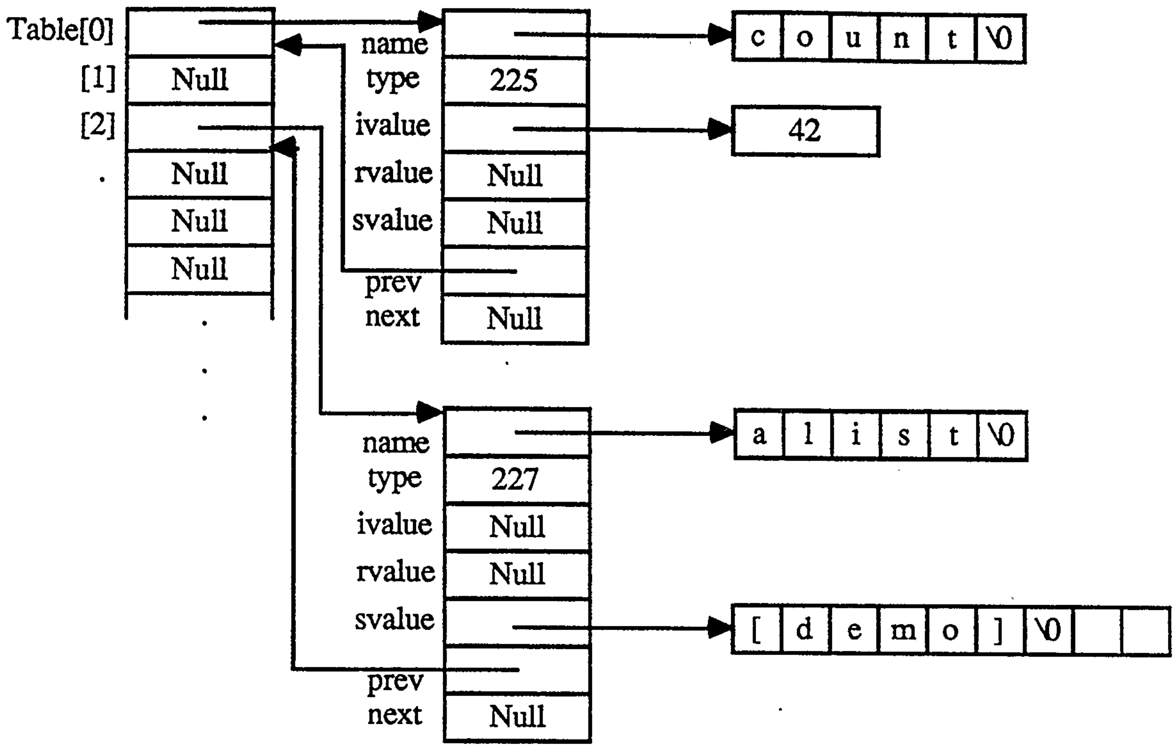


Fig. 9 Data Structure for Storing Variables

The chaining is used in such a way that any insertions involving variables with an identical name to a variable already existing will be placed before the existing variable in the chain. A new variable will be added when the variable name is not found when a *make* statement is encountered, when local variables are defined, and when parameters are defined within a procedure. A list of pointers is used to keep track of any parameters or locals (but not globals added by use of the *make* statement) added by the procedure. When the procedure ends, the locals and parameters used by the procedure are deleted and the variable table is left the way it was when the procedure was called. In this way, there is no need to use a stack to keep track of variables within a procedure; the features of the variable data structure are used as a stack instead. This reduces the load on the main stack.

5.3.3. Storage of Programs

Programs are stored as entries in a table of programs. A hash table is not currently used for this. A maximum of 255 programs can be defined at once.

Each entry in the table (fig. 10) is a pointer to a structure consisting of the name of the program, the text of the program, and the type of program it is (i.e. whether it is defining a sprite or not).

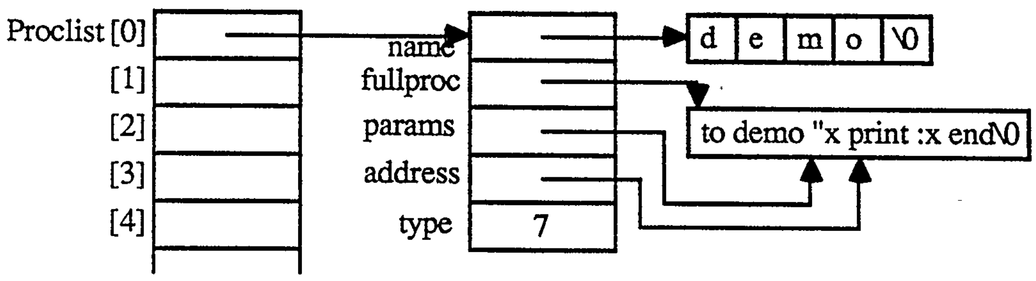


Fig. 10 Data Structure for Storing Programs

When an unknown identifier is found, the table of programs is searched to see if the identifier is a program name. If it is a program name, the parameters to the program are given their initial values. Then, input is taken from the text of the program until the END statement is reached. All the housekeeping regarding the stack is carried out.

Error checking is done to check if too many programs have been defined, and an error message printed if there is no more storage in the table.

5.3.4. The Stack

Since MOGL is a recursive language in its design, the stack is a very important part of the interpreter.

The stack is simply an array of pointers. Pointers are placed on and removed from the stack using the push() and pop() procedures.

Everything that is pushed on the stack must be given as a pointer. Due to the type flexibility of C, the pointer could be a pointer to any type of data; it is the responsibility of the procedure using the stack to keep these types straight.

Error checking is done to check for an overflow.

5.3.5. Other Minor Data Structures

Other data structures are required within MOGL for some minor tasks.

For example, the mirrors are stored as an array of pointers to structures representing the mirrors. Each structure contains an angle that the mirror is turned, and the x and y co-ordinates of the centre of the mirrorline.

5.4. Interpreting MOGL commands

A cross-section of the MOGL commands will be described to indicate how the interpreter was implemented to allow these commands.

5.4.1. The *line* Command

This command calls up the line sprite. This sprite is the basic building block for all other sprites.

The *line* command first erases the old sprite by drawing it (all sprites are drawn using Exclusive-Or on each of the points in the sprite. Thus, if a sprite is drawn once, it will appear. If it is then drawn in the same place again, it will be erased). The pointers used in the previous sprite are deleted. Space is obtained for a single line, and the end-points of the line are set to (0,0) and (1,0) in the conceptual plane to make a line one unit long lying along the x-axis.

The line is then drawn, so it appears on the screen.

5.4.2. Transformations

The procedures that do the various transformations to the sprite first read in any values required as arguments. Then, the transformation is done by calling the exmatrix() procedure and passing as parameters the appropriate matrix. All values used here are double precision reals, to ensure maximum accuracy.

The transformation procedure works by first drawing the current sprite (thus erasing it). Then, each point in the list representing the sprite is transformed by the appropriate matrix. Once all points have been updated, the sprite is drawn again (so it appears in the new location).

5.4.3. Making New Sprites

When a new sprite is being created by a program, pointers are kept to the beginning and end of the sprite being created. Each time a *mark* command is encountered, the list representing the currently active sprite is appended to the end of the list representing the sprite being created.

If a line overlaps another line exactly, and the sprite is drawn, the line would disappear completely; this is due to the exclusive-or technique of drawing points. So, each of the lines in the newly appended list are compared to the lines already in the sprite, to check for overlaps. If any lines are found that completely overlap each other, one is deleted.

Normally, each line is drawn on the screen as the sprite is being created, so that the user can see the shape being created on the screen. However, this may be tedious for large sprites that are created recursively. The user would have to watch each smaller sprite being created separately, and how these fit into the larger sprite. To prevent the input/output intensive drawing of each line individually, a *noview* command is available. If this command is given in direct mode, each sprite will simply appear after it has been created. The internal processing is the same, but only the final sprite is drawn on the screen. The default mode can later be restored using the *view* command.

5.4.4. Flow of Control Commands

MOGL has two commands controlling the flow of statements (other than recursion); the *if* command and the *repeat* command.

The *if* statement is interpreted by first evaluating the expression, by calling the getsymb() procedure.

Next, the first list is read in using getsymb(), and a pointer stored to the text contained in the list. If the character after the list is an opening square bracket, the second list is read in, and a second pointer points to it. If there is no second list, this pointer is set to NULL.

Finally, if the condition was TRUE, then the first list is executed by pushing the current text input position on the stack, and then setting the text input to come from the list. Control would then return to the top level of the interpreter.

If the condition was FALSE, the second list is executed in the same manner (if it exists).

Once the interpreter has reached the end of the list, the old input position will be popped off the stack, and interpretation will resume normally.

The *repeat* statement is interpreted in much the same way as the *if* statement.

First, the expression is evaluated to see how many times it is to repeat. Next, the list of commands to repeat is read in using getsymb(), and a pointer kept to the location of this list.

To actually execute the commands repeatedly, first the old input position is pushed on the stack. Input is then set to come from the list returned by getsymb(). The number of times to repeat is pushed on the stack.

Control is returned to the top level of the interpreter. When the end of the list is reached, the interpreter will pop the number of times to repeat off the stack, decrement it, and check if it is less than 0. If not, it is pushed on the stack again, and the list is interpreted again. This process continues until the commands in the list have been interpreted the required number of times.

5.4.5. Expressions

If the getsymb() procedure finds a value (either a constant or a variable), and a mathematical operator follows that value, then an expression has to be evaluated to come up with the final number. This is done by calling a procedure called readexp(), passing it the type of value read in so far (the actual value is stored in a global variable).

If the getsymb() procedure finds a bracket or negative sign, it also assumes that an expression has been entered. A recursive call to getsymb() is made to evaluate the contents of the brackets or the number to be negated.

The readexp() procedure evaluates the expression recursively. It first initializes a global flag to keep track of errors. This is necessary because the procedures will be returning numerical values, so an error flag can't be returned by the procedures (as elsewhere in the program). Once the initialization is done, the value of the expression is determined by calling readexp1(). The **type** of value is then returned by the readexp() procedure to the main interpreter, and the **actual** value is placed in a global.

The readexp1() procedure deals with boolean operators. It evaluates the expression by calling readexp2(), and then checking for the compound operators <, >, =, <=, =<, >=, =>, <>, and ><. If no boolean operator is found, the value returned by readexp2() is returned directly. If an operator is found, readexp2() is called again to get the second half of the expression, and the result of applying the boolean operator is determined and returned.

The readexp2() procedure works in a similar way, except recognizing and evaluating the addition and subtraction operators. Any number of values can be added together, so it calls readexp3() as long as addition and subtraction operators are encountered; keeping track of the overall sum.

The readexp3() procedure deals with multiplication and division. Any values used by readexp3() are found by calling the getsymb() procedure. Thus, the hierarchy of operators is maintained, as well as bracketing and negation.

The type returned by the readexp() procedures depends on the type of operation done. If a boolean expression is evaluated, the type boolean is returned. If integers are added, subtracted, or multiplied, then the type integer is returned. Otherwise the type returned is real, because at least one real number is used in the calculation, or a division was done.

The relational operators = and <>, when applied to reals, check that the two reals are equal or unequal according to a tolerance that is defined in the program. If two reals are within 0.000001 of each other, they are considered to be the same.

5.4.6. Graphics

One module of the program deals with the details of the graphics. If the program were to be used on a different terminal or a microcomputer, this would need to be rewritten for the new hardware.

A program, tek.c, consists of driver routines to send the control sequences to the Visual 550 terminal, saving some of the tedious details of developing a graphics interface from scratch.

The graphics section of the MOGL interpreter has procedures which call the appropriate tek.c procedures to initialize the graphics, clear the graphics and text screens, and draw lines.

The MOGL interpreter includes a line-clipping procedure to make sure that all lines will fit within the screen or window being used (Newman and Sproull, 66-67).

The graphics in MOGL is a three-layered approach, similar to that discussed in van de Bos (95). The top layer is device independent, using an imaginary plane with the co-ordinate (0,0) in the centre and with units approximately a centimeter long. The middle layer is a simulation layer. Here, co-ordinates in the top layer are converted to those appropriate for the terminal being used. Measurements are converted to pixels, and the plane converted to that used by the terminal (e.g. with (0,0) in the bottom left corner for the Visual 550). All these conversions are done by function calls; the functions would need to be rewritten if the program was used with graphics device using a different co-ordinate system than the Visual 550. The bottom layer consists of the basic device driver, containing the necessary commands to draw lines on the given device. Naturally, the bottom layer would need to be completely rewritten if MOGL were used on a different machine.

5.4.7. Variables

Variables implemented in MOGL are set up so that the user does not need to be overly concerned about how the variables operate. Thus, variables do not need to be declared before a value is assigned to them, and the type of a variable does not need to be defined in advance. A variable must have a value before it is used in an expression, however.

If the *make* statement is used to give a value to a variable that doesn't yet exist, an entry is made in the variable table to make a variable of the given name, and to assign to it the type as reflected by the second parameter to the *make* statement.

If a subsequent *make* command is used to assign a value to the variable, the old value and type are deleted (and the space they used is freed), and new space allocated for the new value. In this way, if a different type of object is assigned to the variable, it is just updated to the new value and type.

The *local* command creates a variable of the name given which is local to the current procedure (i.e. it is added on to the beginning of the variable chain and to the list of variables to be deleted at the end of the procedure). This is done by creating a new variable with no value, and adding it to the beginning of the variable chain and to the list of variables which are to be deleted (see section 5.3.2.).

Whenever a variable is found preceded by a colon, the interpreter will replace it with its value before continuing on the interpretation. If the variable has no value, or doesn't exist, an error message is printed.

5.4.8. Communication with the User Interface

The interpreter for MOGL can work in two modes; with a user interface, and without it. MOGL can operate without the interface, if desired, by calling it with a "-ni" flag. If used without the user interface, MOGL will simply print any text directly on the screen, and draw graphics behind the text. Both text and graphics can take up the full screen.

If a user interface is attached, however, the interpreter must be able to provide the interface with the appropriate information, and receive commands from the interface.

A boolean global, *interface*, is true if a user interface is attached. If an interface is attached, the input and output routines in MOGL must accommodate the interface.

Communication between MOGL and the interface is attained by having MOGL call various procedures in the interface module, and vice versa.

For example, when the interface is in effect, MOGL may only have a limited window to produce graphics on. Thus, special line clipping must be done. So, if an interface is attached, MOGL will first call a procedure gwindsize() to find the coordinates of the graphics window. All future drawing is done within those confines.

MOGL will also send textual output to appropriate procedures in the interface to print the messages in an appropriate place. Error messages, status messages, and output that the user has requested will all be passed to different procedures for processing.

The interface for MOGL requires that the screen be destroyed when the LEARN mode is entered, and later redrawn. Thus, all the *mark*s done in direct mode must be remembered. This is done by creating a sprite-like structure, containing marks. When necessary, all the marks done so far can be re-drawn.

MOGL is designed to be able to receive text from the interface as if it had been typed. When MOGL is expecting input from the user with the interface in place, a procedure in the interface module will be called to return the appropriate text. MOGL will then act on the text as if it had been typed by the user.

By designing the interaction between MOGL and the interface in this way, they remain relatively independent; with only a few procedure calls required to transfer control between the two modules.

5.5. Structure of the User Interface Module

The user interface module for MOGL has to have many capabilities to be able to handle some windowing operations, as well as allowing for pointing device input and graphical output.

It would be easiest to write the interface using an existing windowing system, but systems such as the Maryland window system don't allow windows containing graphics.

Thus, a unique windowing system had to be designed. Some information on this task is available (Rochkind) but information on graphical or pointing device operations is not included.

Graphics are limited to a pre-defined rectangle on the screen. Thus, there is no need to worry about advanced windowing operations such as the capability to move windows. The interpreter is told the confines of the graphics screen, and given new coordinates if the screen is moved (if the LEARN mode is entered, for example).

No provision is available for dealing with the pointing device input on the visual 550 terminals, so that is custom designed as well. The user can choose to use the arrow keys or the regular text keys. If an arrow key is pressed, the cross hairs move in the appropriate direction (this capability is built in to the visual 550 terminals). When the space bar is pressed, the coordinates of the crosshairs from the terminal are passed to a procedure which handles the menus. This part of the program is not portable to other terminals, and will need rewriting. If the user types alphabetic characters, they are stored and printed out within the confines of the text window. As soon as a string has been entered it is passed directly back to the MOGL interpreter for processing.

All window coordinates throughout the interface module are given as textual x and y coordinates, indicating the position of characters on the screen. Functions are used to convert these to absolute graphical coordinates for the drawing of appropriate lines.

The windows are stored as an array of records. Each window has a name (which appears at the top), x and y coordinates of its top left-hand corner, a width and a length, a position of the text cursor within the window, and the textual contents (if appropriate).

The menus for the menu window are also stored as an array of records. Each menu has a name, and a list of its submenus (anywhere from 0 to 8 submenus are possible). Thus, if the MAIN menu is displayed, the names of all the submenus are printed. If, for example, the clearscreen submenu is selected, it has no submenus, so the command "clearscreen" is directly passed to the MOGL interpreter. If there are submenus, the menu structure is traversed until the end is reached, with the necessary pieces of the commands being assembled as the structure is followed, and the complete command is passed to the MOGL interpreter at the end.

Some menus are special cases (such as the LEARN menu which requires a new screen to be created, or the SPRITE menu which needs to get the current list of sprite names from the interpreter). These menus are indicated by having a negative number of submenus. The negative number tells the interface to use a special procedure.

Some submenus are special cases as well. If a negative number occurs in the list of submenus, this means the submenu is a dial or a scale. If the number is -1, the submenu is a dial; if it is a lesser number (-2 or less), a scale is made with a range anywhere from 2 units up (according to the number used). The numbers selected from the dial or the scale are printed, and included in the string that is to be passed to the MOGL interpreter.

Several special printing procedures are used in the interface, so that messages are printed in the appropriate windows. Commands are printed in the TEXT window as they are entered; any textual output from the commands is also printed there. Error messages and status messages are printed in the MESSAGES window. All of the printing procedures call a common text handling procedure, which makes sure the text wraps properly within the window, and handles scrolling if the window is full.

The user interface is conceptually quite simple. However, the procedures to draw the windows and the menus, as well as deal with crosshair selections made by the user to traverse the menu structure, can be quite complex. Many support procedures had to be written (such as those to draw the dial and scale), and custom input and output routines were written to ignore any characters not within the character set used by MOGL. All of this has led to a relatively robust and easy to use user interface.

5.6. Conclusion

The programming of MOGL was an involved process. It included using matrices for transformations, dealing with both typed and programmed input, processing errors, creating and implementing data structures for many kinds of objects, and interpreting the individual commands. The graphics portion of the language used a three layer approach for simplicity. As well, an entire windowing system had to be designed. The end result is a prototype of the MOGL interpreter, with enough functionality to get a good idea of how the language works, and to do some preliminary testing on the use of MOGL.

Chapter 6

Evaluation and Discussion

6.1. Introduction

The evaluation of MOGL will be attempted by looking at several topics. These are as follows:

(a) The user response to MOGL.

(b) A comparison with another microworld - an implementation of LOGO - will be made.

(c) Comparison of the motion geometry features of the MOGL language to the design goals.

(d) Comparison of the user model implied in the design of MOGL to theory, to see how well it meets the needs of users and learners.

(e) An analysis of the user interface to see how well it meets the various criteria for user interfaces. And finally,

(f) A discussion of the actual implementation, to discover possible programming improvements.

6.2. A survey of users

User response to MOGL was determined by surveying a group of six users who were taught various commands in MOGL and asked to make comments on how the language worked. The users consisted of both males and females from 16 to 27 years old with many different levels of computer experience ranging from near-novices to experts. As this is a prototype version of MOGL, this should be a sufficient survey. The comments these users made will indicate areas to look for improvements in the MOGL language and user interface.

The survey (Appendix A) was given to six people, and allowed them to use the turn, slide, and reflect commands, as well as letting them define a new sprite.

The users came up with many useful comments, and discovered several shortcomings of the current implementation of MOGL. Each user issue will be addressed individually, and at the end of this section the user suggestions will be summarized.

The user comments are as follows.

Some users had difficulty with the term "sprite" for the shape being moved on the screen, suggesting it should be called a "shape" instead. This can be easily changed in the interface.

When moving the sprite around the screen, and leaving marks, users found that the mark can be confusing; whenever a mark was left behind, the sprite could no longer be distinguished (this was explained in the survey by saying the sprite was behind the mark). It would be better if the sprite stayed on top of the footprints. However, this would present a programming problem, as the sprite is currently drawn using an exclusive-or technique. This problem could be solved by using two graphics layers, one containing the marks and the other containing the sprite. If the sprite is on the top layer, the problem would be solved. The Apple][computer (used in the Calgary school system) does have two graphics screens available, so this would be reasonably easy to program. The Visual 550 terminal does not have this capability, so to program this type of feature would require computing the intersection points between lines in the sprite and the footprints. This could slow the program down somewhat, but would solve the problem by ensuring that lines that overlap are drawn correctly.

Other confusion was caused by the positioning of the line sprite initially; users wondered why it was the length given, and positioned with one end on the centre of the screen. Sometimes, users had difficulty telling the position of the sprite on the screen in relation to the centre. These problems might be solved by having an option to make the X and Y-axis visible in the background, possibly with markings for each unit of length. This may have to be experimented with, to ensure the screen does not become too cluttered.

Using the *turn* command caused some confusion. Because a dial was used to indicate how far the line sprite was to turn, users thought that it was to give the desired orientation of the sprite. It was not obvious that this was a turn relative to the current position of the sprite. There were other problems, with users unsure about which direction the sprite would turn. One way to solve this problem would be to animate the turns; showing the sprite every 5 degrees or so as the turn is done. In this way, the direction of turning would be obvious, and the total distance turned easier to visualize. Although for a line sprite it might seem simpler to position the dial to fit the original orientation of the sprite, some other shapes (such as a square) do not have an obvious orientation.

Users were confused about which point on the screen defined the centre of rotation for the sprite. This could be simplified by having the X and Y-axis visible. Some users would prefer the sprite to turn on its own axis. This would violate the motion geometry concepts being taught, in that the sprite should be seen to rotate around a point on the plane rather than its own centre. An advanced user, however, could write a turn procedure which would turn the sprite around an arbitrary point, and learn quite a lot about motion geometry in the process.

One user found that the dial used in the *turn* command was difficult to get used to at first, having trouble in selecting a position on the dial and then selecting *accept* to accept the position. Using a mouse might solve this problem, with the concept of dragging a pointer around the dial easier to visualize.

In comparison, the *slide* command caused few problems. The menus came up in the order the arguments were required, and the dial was used for the absolute heading. Both of these things made the slide command straightforward to use, and users commented on its ease of use.

Some users expected the *clearscreen* command to clear the entire screen, rather than just the graphics window. The command can be renamed to *cleargraphics* instead.

Reflections caused a little confusion. One problem was that the users expected the mirror to act like a real-life mirror; showing an image of the sprite rather than moving the sprite to a different location. Some left a mark before reflecting to show both the original and the reflected position. This was a good way to visualize the reflection transformation, and could be used from the beginning. It should not be available automatically, though, since reflection is a movement in motion geometry, not a doubling of a shape.

When more than one mirror was on the screen, there was some confusion as to which mirror was which. This problem could be solved by having a number visible beside each mirror. This is not really a suprising problem, and the numbering of the mirrors was not implemented because this is only a prototype implementation.

Users generally enjoyed creating a new sprite, using words like "fun" and "neato" to describe this experience. This part of MOGL seemed to be very successful, and users often tried creating several sprites enthusiastically.

One source of confusion occurs in creating new sprites. In the direct mode, the *mark* command would leave a permanent "footprint" on the screen, but in the sprite program mode, it defines a part of the new sprite. This confusion could be removed by explaining this difference to the user before programming is attempted (this should have been done in the survey). The feature of the mark command working in a different way when programming allows users to expand on concepts previously learned, so it does help the users in the learning process.

Another problem is that one user thought he had crashed the program when selecting *LEARN* because it took a long time to draw the *LEARN* screen. Perhaps having a clock on the screen during long operations would help.

Two of the subjects made errors while typing in their programs, and were unable to correct these errors. A text editor would be helpful for fixing any mistakes made in programming.

In loading sprites, one user mentioned it would be nice to have a directory command to see what sprites are available. This would be straightforward to implement, requiring a bit of interfacing with the operating system.

When typing commands, one user found that using a negative number as a second argument to a command (e.g. -5), a formula would be evaluated (such as 4 - 5) rather than using the two numbers separately (4 and -5). The problem is caused by the arguments to commands needing to be separated only by spaces. This could be rectified by warning users to type negative numbers inside brackets, although this isn't necessarily a good solution. Using a comma as a delimiter between arguments would also solve this problem, although this would make MOGL different from Logo.

The users gave many comments about the user interface, and quite a few improvements will have to be made in this area.

A user commented that he would like to see commands that have scrolled off the screen. This could be rectified by logging all the commands, and using a scroll bar to retrieve old commands.

Users found that some error messages were not helpful. These error messages can easily be re-written to be more descriptive.

Most users liked the option of being able to type or select commands, and used both methods throughout. This integration of the beginner and expert users seems to be enjoyed, especially by experts learning the system. One mentioned that he thought the menus were a good way to learn the commands, and would then type them after using the menus a couple of times.

Almost all the users found typing commands in easier than using the crosshairs, but most thought a mouse would improve the selection process. One user mentioned that he would prefer typing over a mouse. The crosshairs are difficult to use, and a mouse interface must be added to MOGL.

Once or twice, a user would start typing, and then be frustrated because the crosshairs would not be available again until the return key was pressed. Perhaps a mode could be used where the menus will follow what is being typed on the keyboard, so the correct menu for a given part of any command is visible while typing is done. This may be distracting to the expert user, though, so should be added as an option for non-experts.

There was some confusion about hitting space to select a command with the crosshairs and hitting return to enter a typed command. This would no longer be a problem if a mouse were used.

Some users did not like the way the menus were set up; preferring instead to have all the commands on one menu (indicating that Miller's 7±2 concepts in short term memory may not be applicable) or at least having the commands on the main menu grouped together, away from the sub-menus. The grouping of the menus could be improved by grouping commands and sub-menus on separate parts of the menu screen. However, having **all** the commands on one menu would make it more difficult to find and choose the desired command, as any sprites that are defined would have to be displayed along with twelve commands.

Some found they went to the MIRRORS... menu when trying to reflect, even though reflect was a movement and under the MOVE... menu. This again is confusion with motion geometry concepts rather then MOGL, as reflection is a movement in motion geometry. In this case, the interface was implemented in the best way possible.

An escape menu to get back from a menu without executing the command would also be an option. This can be done in the current implementation by selecting an empty menu slot; but this produces an error message. An escape command can be added to the bottom of every menu, instead.

Several users mentioned that an undo command would be useful when mistakes are made in the drawing. This command could be added to the main menu, so the last entire operation is undone. Since all the operations in MOGL have the characteristics of a group (as defined by Piaget), any action can be undone by executing its inverse. For example, to undo a rotation, another rotation could be done by the negative of the original turn. The specific inverses for each command would need to be programmed as part of the undo command.

One user overloaded the one character keyboard buffer several times; allowances will have to be made for fast typists. This modification will be difficult, since the UNIX buffers can't be used. A buffer will have to be added to the interface, with periodic checks to make sure nothing has been typed every time a long operation is done.

Generally, the people using the program seemed to enjoy working on the computer. They found MOGL "friendly." It was interesting to notice that while doing the survey, three of the subjects completely forgot about the survey, and started experimenting with MOGL on their own and creating new and interesting shapes on the screen. This was not discouraged, as the whole point of MOGL is that it is supposed to be a microworld; a place to explore. All the subjects were quite complimentary in rating MOGL, even with the flaws that they noticed.

MOGL seems to have achieved the main goals for the system. Users found it fun to use, so were highly motivated to explore the MOGL microworld. The motion geometry commands worked as expected most of the time, and when an unexpected effect was discovered, it was due to a misunderstanding of motion geometry on the users part rather than an inconsistency in the system.

Many extensions to this prototype of MOGL are suggested by the experimentation. The X and Y axis should be made visible. The turn command could be animated. A mouse must be added to the interface. A clock should indicate long operations taking place. A directory command would be useful. Having the menus optionally follow the typed commands could help intermediate users. And finally, a keyboard buffer should be added. These were issues not addressed in the design that should be dealt with in a future re-design.

Some of the design features were not implemented as this is a prototype. Some of the many minor features were omitted. Most notable of these are the escape command (to go back a menu) and the undo command (to undo an operation). As well, the mirrors should be numbered to indicate which mirror is which. The error messages are not as helpful as they could be. And finally, the marks could be rewritten so as not to hide the sprite (although this could be difficult to program).

Some of the decisions made in the design process were in error, and now could be changed. The word "sprite" should be changed to "shape," for a more natural way to talk about objects on the screen. The *clearscreen* command is misleading; *cleargraphics* would better indicate that only the graphics screen is cleared. A *directory* command would help the user find available MOGL programs before loading. Some way of logging commands that scroll off the top of the screen would be good as well, so users could check on old work when doing new operations.

The menus were grouped so that the user would follow the menus in order in a typical session; starting with the top menu and working down through the menus on the main menu. Thus, a sprite could be defined, moved, and then made to leave a mark behind. Users would prefer to have commands grouped so menus are separated from commands that execute immediately.

6.3. Comparison of MOGL and Logo

The basic format of commands in MOGL is based on Logo, to make it easier for students who have learned one microworld to quickly learn the other. However, there are some differences between MOGL and Logo (besides the different types of geometry used). MOGL will be compared to Apple Logo, which is used widely in the public schools.

6.3.1. Possible Additions to MOGL's Commands

MOGL has no facility for accepting any kind of input into variables from the user of a program other than using parameters when calling a procedure. It would be good to implement commands like Logo's *readlist* or *readword* commands to read information in while a program is operating. Users could use this in programs to define shapes where questions must be asked while the shape is being defined. As well, teachers may need this facility to create programs for the students to use. Other commands would be necessary with the input commands, to break lists down into words and convert words to integers or reals.

An operation that can be quite frustrating in MOGL is the *save* command. This command saves all of the procedures currently defined. However, the user may only want to save one procedure. Logo uses a command called *savel* to save only those procedures named in a list given as a parameter to the command.

There are many other commands in Logo that may be useful to MOGL, but these two major commands are the only ones needed to make MOGL more useful as a motion geometry microworld.

6.3.2. Variable Names

Variables are passed between MOGL procedures in two ways. First, they can be passed by value (only the variable's value is passed), or they can be passed by name (the name of a variable is passed). There is no facility to pass variables by reference. This is similar to the way that Logo handles variables, but can be confusing in recursive procedures to a programmer used to passing values by reference. It may be desirable to add a capability of referring to variables by reference in MOGL.

MOGL does have a local command to make variables local to the procedure in which they are defined. Otherwise, all variables defined anywhere are global. This is the same as Logo.

One difference in the use of variables between Logo and MOGL is that when a procedure is defined, MOGL requires the names of any parameters at the beginning of the procedure. For example,

```
to demo "n1 "n2
```

This was designed in this way because it makes sense to think of a procedure being defined along with the *names* of its variables.

Logo requires procedures to be defined using the symbols for the *contents* of variables (using a colon rather than a quotation mark). For example,

```
to demo :n1 :n2
```

This is not as logical as the way MOGL is written. However, MOGL might be re-written so that procedures are written in the same way as Logo, to avoid confusion when a young user switches between the two languages. This small logical inconsistency is outweighed by avoiding confusing the user with a new format of writing procedures.

6.3.3. Different Objects in MOGL

In Logo, all objects are implemented as text strings. Thus, even the orientation and position of the turtle is available to the user through special commands, returning a list containing the coordinates and orientation. This allows consistency with Logo as a complete programming environment.

MOGL, however, uses a special data structure to represent sprites. Because speed is important, it would be inefficient to represent it as a text string. Since MOGL is really an environment for exploring motion geometry rather than exploring general programming, this lack of information about the sprite is unimportant, and more than compensated for by faster speed than would be realized with a text representation.

6.4. MOGL's Motion Geometry Commands

The main emphasis in designing the command set for MOGL has been the creation of a number of motion geometry commands. However, now that the language is implemented, it is apparent that some of the commands could be improved.

In using MOGL, it would be reasonable to allow default parameters for some commands to increase simplicity for novice users. For example, if no distance is given to the slide command, it could slide the shape one unit in the direction given. Unfortunately, MOGL is written to allow several commands on one line, and it would be difficult to parse the slide command if it could have a variable number of parameters. A more reasonable approach might be to create a library of default commands which could be loaded as MOGL procedures. These commands could have the desired number of parameters, and operate in a default way. For example, to create a command called *sl* which would slide the sprite one unit in the direction indicated, the following program could be used:

```
to sl "angle
        slide 1 :angle
end
```

Another difficulty with MOGL is that the enlarge command can unexpectedly cause a sprite to disappear. Since the sprite is being enlarged around the centre of the screen, if it is off-centre and enlarged too far, it will seem to disappear. This feature could be explained to the user by having him or her imagine the sprite as drawn on a large balloon, where the screen looks at just a part of the balloon. If the balloon is blown up too far, the sprite will no longer be visible on the window; but it still exists off the screen.

The enlarge command was written to be a simple linear enlargement, as this is what is covered in the motion geometry curriculum at the outset. If a user desires, it would be possible for him or her to use MOGL to write a more general enlarge command, which enlarges around an arbitrary point on the screen; or enlarges only in the horizontal or vertical direction.

The *turn* command caused quite a few problems in the survey given to various potential users. The *turn* command was designed to turn in only one direction for simplicity. In an early design, it was decided to have a *turncw* command and a *turnccw* command which turn clockwise and counterclockwise, respectively. Some potential users expressed confusion with these commands, so they were omitted. Users can instead use a negative number with the turn command to turn the sprite backwards.

Many users complained that MOGL always turns, reduces and enlarges around the centre of the screen. To address these concerns, MOGL could be given a function to return the centre of a shape. Using this, an experienced user could write custom turn, enlarge, and reduce commands which all operate around the centre of the shape, and in the process learn a lot about how motion geometry operates.

6.5. Does MOGL Meet the Educational Goals?

MOGL appears to be a successful microworld. It is a self-contained universe that is essentially consistent and can be explored quite readily. With the addition of the user interface, this exploration is quite easy. In fact, some of the subjects in the survey completely forgot they were doing a survey and started exploring MOGL on their own. They would get quite excited, and start experimenting with different menus and creating different shapes. The subjects were not discouraged from doing this, because MOGL is designed to be explored. MOGL shouldn't really be used according to a worksheet.

Because of the motivation that users seem to have while using MOGL, and the way MOGL works consistently with established principles of motion geometry, it seems that it will be easy to get students to use it. As well, since it was designed with current educational theories in mind, MOGL should be an effective educational medium.

6.6. The User Interface

To do an analysis of how effective MOGL's user interface is, it is useful to refer to some of Hill's 13 design goals (Hill, 6-12).

1. Know the user

Research was done into the ways that people learn, and this has been applied to the design of MOGL. Since it is designed as an interface for school aged children and older people, there are various ways of interacting with the system. A pointing device can be used, and direct typing of commands is also possible.

2. Design the tools the user needs

MOGL allows the user to use all of the major motion geometry operations. Pictures could be drawn in a more straightforward way (such as pointing to a location to slide to rather than giving a distance and direction to the slide command), but this would not allow the user to really understand the underlying concepts of motion geometry. The emphasis throughout has been to design an educationally viable environment for learning these concepts.

An omission from the current MOGL system is a text editor. At the moment, if the user makes a mistake entering a program, he or she must save the program, leave the MOGL environment and use one of the Unix editors. This is unacceptable. A good editor should be included as part of the MOGL environment, perhaps within the TEXT window.

3. Make the system easy to learn and remember

The interface to MOGL is designed as a menu structure, making it quite easy to learn the commands, as users have commented. The words used for the commands usually have obvious meanings. The structure of the menu as a hierarchy follow the way humans may structure the concepts used as a mental hierarchy (as described by Piaget).

4. Avoid both human and machine failure modes.

Comprehensive error checking is done throughout MOGL to make sure that any errors won't cause the program to crash. However, the break and delete keys can cause the program to stop. In the experiments, this problem was solved by physically disabling those keys. A software solution to this problem would be preferable. This requires rewriting some of the input/output routines for MOGL.

To avoid human failure, the system is designed to be reasonably friendly, and commands are available to do repetitive tasks (using recursion or the repeat command).

5. Structure the interaction

MOGL's interface is highly structured, allowing the novice user to follow the menu hierarchy to execute commands. However, the structure could be better represented to the user. All the menus will appear in the same place on the main screen, no matter how deep the user is in the structure. It might be better to do something similar to the overlapping done by the LEARN screen, and overlap any new menus slightly. In this way, the user could see how deep she or he is in the menu structure.

6. Convey a feeling of control to the user

The users tested in the survey indicated that they did feel quite a lot of control over the system. Having the menus meant not having to guess at commands, and being able to execute desired actions without too much trouble.

7. Consider division of labour

The computer should only do what it is suited to do, and the user what he or she is to do. Thus, the computer should be dealing with almost everything not concerned with the learning of motion geometry.

Rather than having the user guess where the sprite is when there are marks on the screen, the computer should do the necessary computations to ensure the sprite is visible at all times.

Otherwise, the MOGL interface seems to take care of most of the details. Some things could be automated, such as the naming of new sprites the user creates, but it is better to give the user some control in these areas.

8. Take into account human performance limits

The interface is set up in a way that allows easy interaction. The main focus of attention is the graphics screen, so that has been placed in the middle.

9. Provide facilities to monitor system activity

There are, as yet, no facilities to monitor activity on MOGL. Many types of monitoring should be implemented. Each error encountered should be logged, so weaknesses of the system can be determined. A facility should be available to record the timing of each selection and keystroke, for future analysis with the keystroke level model (Card and Newell). And a gripe facility might be a good idea, so users can offer complaints and suggestions for the system.

Perhaps the first improvement to MOGL should be to add a mouse to the system. A difficulty would be having the program communicate with the mouse, but languages have been designed to help deal with this problem (Cardelli and Pike). Using these techniques, it should be relatively easy to add the mouse capabilities to MOGL.

6.7. Implementation of MOGL

There are some considerations in the implementation of MOGL that should be addressed.

The implementation of all the procedures so that they return a value indicating if an error has occurred seems to be quite effective. Thus, if a single error does occur, the user can be returned to the top level without any major problems. Most of the error messages are effective, but some must be improved (for example, if the user selects an empty slot on the menu, the error message received is "slot unbound"). As well, better garbage collection must be implemented, as MOGL has a tendency to run out of data space when heavily recursive programs are run.

The implementation of MOGL in C seems to work well. It is portable; only requiring re-writing of the graphics module and some of the text input filters. The three-layered approach to the design of the graphics system should make the redesign of the graphics system fairly straightforward.

There are many minor improvements which should be made to the program.

Variable names are stored in a hash table with chaining, but program names are not. This means that although the number of variables available is limited only by memory available, the number of programs that can be defined at once is only 256. And since the program list must be linearly searched each time a program is called (an O(n) operation), recursive programs are slower than necessary.

When a sprite is defined, each line that is added to the sprite is checked to make sure there is no overlap. This is designed so that lines won't mysteriously disappear when two lines are in the same location, since the sprite is drawn using an exclusive-or technique. However, it is a very slow operation ($O(n^2)$), depending on the number of lines already defined. Another problem with drawing the sprite using this exclusive-or technique is that if the sprite is drawn over top of a mark, it seems to disappear behind the mark. A final problem with drawing the sprites with the exclusive-or technique is that if the sprite is to leave a mark, it must first be re-drawn (to erase it), the mark drawn in the shape of the sprite, and the sprite re-drawn again (to make it reappear). This can take a long time for larger sprites.

It might be best to draw the sprites directly as solid lines, perhaps keeping track of what the sprite covers. Then, when the sprite is removed, the objects the sprite had covered could simply be redrawn. This would mean the sprite always appears as made up of solid lines, and the checking for overlapping lines within the sprite would be unnecessary. However, more overhead is brought into play by having to somehow keep track of what the sprite covers.

MOGL needs additional programming to ensure it is absolutely "bombproof." At the moment, it will crash if it runs out of memory, or if the delete or break key is pressed. The memory problem can be fixed by having more stringent error checking for space, and producing an "out of memory" error message which allows the user to at least save the workspace. The problems with mistaken keypresses can be fixed by removing the filters that the Unix operating system puts each keystroke through before passing it to the program (this would also increase MOGL's portability). Apart from these three problems, MOGL appears to be relatively secure.

As well as the problems noted above, the program requires some extensions to allow a more complete command set for MOGL. No major problems are foreseen with this process.

6.8. Summary

The MOGL language and interface seem to be quite effective. In the process of surveying users, it was found that they had fun using the current version of MOGL, and generally enjoyed learning motion geometry. However, some improvements and modifications can be made to MOGL.

The user survey indicated that MOGL should be completed according to the design, including a mouse interface and commands to skip menus and undo operations. As well, some additional features will have to be added to MOGL that were unanticipated in the design phase. These include many extra features (such as optional display of axes) to make the user more comfortable using the system.

MOGL compares favourably to the microworld Logo, when it is considered that MOGL is only an environment for using motion geometry, and not a complete programming language.

The motion geometry commands are complete and consistent in MOGL, and appear to require no additions or modifications.

The user interface could have several improvements, including the addition of an editor, and a better visual structuring of the layers of menus.

The implementation of MOGL appears to be quite robust, and copes with errors in a reasonable manner. It is not yet completely bomb-proof, and requires some additional programming effort to ensure its security.

Overall, MOGL appears to be a general success with some relatively minor problems. Most of the design goals were achieved, and the final product appears to be a useful microworld for learning motion geometry.

Chapter 7

Conclusion

7.1. Introduction

This chapter will summarize the development of the MOGL sub-system in several sections, as follows:

(a) The problem to be solved

(b) Related Literature

(c) Design Considerations

(d) Implementation

(e) Findings

(f) Justification

(g) Future Work to be Done

7.2. The Problem to be Solved

MOGL has been designed as a computer subsystem, to provide an environment for users to explore concepts of motion geometry. Motion geometry has been taught in the past by having students compute formulae to transform each point of a shape on paper, having students use various hands-on techniques, and with limited computer environments (such as MOTIONS). MOGL has been designed to provide a better environment for learning motion geometry than is already available, including the use of all the important motion geometry commands and allowing arbitrary shapes to be defined; as well as including the best from the other environments.

This new environment was to be designed by applying principles from many different areas of computer science. The investigations for this thesis included looking at other motion geometry systems, human-computer interaction, and some aspects of learning theory.

All of these theories would be used to design a system for motion geometry.

To ensure that these techniques did indeed produce a useful system design, a prototype was to be implemented, and this prototype used by several users.

7.3. Related Literature

Research has been done into many areas to determine the requirements for such an environment, by searching the related literature.

First, another motion geometry language, GROPER, was explored. This provided the idea of creating pictures by moving lines with motion geometry, and creating pictures using programs. As well, it used various motion geometry commands; but not including reflection.

Second, microworlds were investigated, and were found to be an effective way to present abstract geometrical concepts to students. Logo (Papert) and MOTIONS (Thompson) were both analyzed, and concepts from both were to be used in MOGL.

Third, methods of human-computer interaction were researched, including user modelling and interface design. It was decided to model how students actually learn, and to create a system to encourage this. The model students have of the system is also important. The design of the interface to MOGL was made by following some of the thirteen principles of interface design suggested by Hill.

Fourth, current educational theory (by Papert and Skemp), was investigated to try and build up a model of how students can learn.

7.4. Design Considerations

The design was built on these concepts to create a microworld with a similar command structure to Logo, allowing a user to manipulate a sprite to create a picture on the screen. The sprite was designed to be moved around the screen by various motion geometry commands (including rotation, translation, enlargement, reduction, and reflection), and could leave a mark behind at any point. A framework for complex work with sprites was built, including creating new sprites, using variables and recursion. A user interface for the language was also included in the design, allowing the use of windows, menus, and pointing device input.

As this design was elaborated, commands to do all the motion geometry operations, control commands, and file commands were added. The capacity for programming in MOGL was included. Finally, the user interface was also designed to allow the selection of these commands from menus, using windows and pointing device input.

7.5. Implementation

The implementation of MOGL was done in C on a Sun computer under the UNIX operating system. MOGL was implemented as an interpreter. Matrices were used for the motion geometry operations, so that any linear transformation could be described by six numbers, allowing for easy extensions of the motion geometry commands. Error processing was well defined, ensuring that most errors would not cause MOGL to crash.

Data structures were designed to deal with the information. The storage of variables and programs was considered, and a stack had to be implemented to cope with the recursion. Those data structures dealing with the storage of sprites were designed to operate efficiently so that sprites could easily be moved around the screen.

The graphics system was designed using a three-layered approach to allow easy portability of the MOGL language, requiring only the bottom layer or two to be rewritten for use on a different graphics system.

A windowing system had to be created from scratch to accommodate the user interface. This included designing a menu system as well as techniques to select numbers using dials and scales.

7.6. Findings

The prototype of MOGL that was implemented was tested by a group of users using a survey to find how the system worked for each user.

MOGL was found to achieve most of the design goals. Users using MOGL found the commands and the interface quite easy to use, and explored motion geometry enthusiastically. The creation of new sprites was especially appreciated, and users enjoyed experimenting with the language.

Some problems were encountered, including the lack of a mouse for pointing device input, the lack of an undo command, and the fact that there was no way to escape from a menu once it is entered.

MOGL was also compared to Logo, and some suggestions made as to extra commands that could be added to MOGL, such as *readlist* to accept input in a program, and *savel* to save only specified programs when saving the environment.

Some motion geometry commands caused a little confusion for the users. However, this initial confusion was often caused by lack of understanding of the motion geometry concepts. After using MOGL for a while, users became used to the way it operated, and further increased their knowledge of motion geometry. Examples of this included confusion about the original image not being visible once a reflection was done, confusion about sprites rotating around the centre of the screen, and confusion about sprites that are enlarged too much disappearing.

The principles of user interface design seem to have been followed for the most part, however when comparing the final version of the MOGL prototype to these principles, some omissions were found, such as the need for a text editor and "bombproofing" of the system.

7.7. Justification

MOGL is an improvement on the ways that motion geometry has been taught in the past, and appears to be a successful implementation of a computer environment.

MOGL certainly improves on some of the hands-on techniques for investigating motion geometry, since it is not necessary to transform each point in a shape by hand, or to use tracing paper and MIRAs to do simple transformations. As well, there is immediate feedback rather than having to go through a process of manually drawing each line. MOGL allows the exploration of motion geometry to occur in a consistent environment, and allows operations such as enlargement which can not be simulated easily and accurately in the "real world." By keeping students from having to worry about co-ordinates and formulae, the underlying concepts of motion geometry can be more easily grasped.

MOGL improves on MOTIONS, the environment for exploring motion geometry. MOGL allows enlargements and reductions, two operations not available in MOTIONS. As well, MOGL allows the user to draw pictures using motion geometry commands, and even create new sprites to use. This encourages the user to explore, and increases interest and motivation on the part of the student. MOGL also is contained within a complete environment, allowing the user to point to commands rather than having to type them, and keeping separate parts of the interaction in different windows on the screen. By basing the design on many areas of computer science as well as using a little learning theory, it seems that many pitfalls have been avoided so that a good system is created.

The implementation of MOGL has been quite successful. The language was programmed with good error-handling techniques, so that errors do not cause the program to crash. A windowing system was successfully designed, and the graphics capabilities of MOGL appear to work very well. One flaw in this area is that the sprites disappear behind the footprints, and due to the speed of the serial line between the computer and the terminal, some shapes can take a while to draw. Generally, however, the data structures work efficiently, and quite complex sprites can be moved with relative ease.

The user response to MOGL has been very positive, with some suggestions as to improvements to the language. Users enjoyed using the language, and were very happy to sit and explore the microworld of motion geometry, hardly realizing they were learning about an area of mathematics all along.

7.8. Future Work

The prototype of MOGL has been successful enough to make the prospect of a full implementation of the language attractive. This implementation should be done on computers that are readily available in the public school environment, allowing for some sort of pointing device input. The recommendations from the analysis can be implemented in this full version of the language.

Once a full version of the language has been implemented, a more thorough analysis should be made of how effectively a user can use the system. A keystroke level analysis (Card and Newell) and an analysis of user complexity (Kieras and Polson) could be done on the system. Perhaps another experiment could be done, comparing versions of the language with and without the user interface, to see if the interface actually does help the users. These experiments are not really appropriate for use on the prototype, since the mouse is not yet implemented, and the full language not available.

It may also be useful to try integrating MOGL with other software. The two languages MOGL and Logo could both be investigated by students, since they have a very similar command structure. This follows Papert's idea of using microworlds to allow the exploration of mathematical concepts rather than the conventional view of curriculum. Other similar environments for other fields could also be created.

7.9. Conclusion

The goal of creating a better computer environment for exploring motion geometry appears to be realized in the design of MOGL. Students do not have to go through tedious calculations to explore motion geometry; instead they can explore it directly by using commands and moving shapes on a computer screen. MOGL outperforms conventional hands-on techniques (such as using tracing paper and MIRAs) in that enlargements and reductions are possible, and all of the motion geometry commands work in the same environment, requiring only different selections for different transformations. MOGL is also a better environment than MOTIONS, because MOGL has been designed as a complete environment; allowing selections from menus, including enlarge and reduce commands, and creating extra incentive for exploration by allowing students to create pictures on the screen and even their own sprites.

Some modifications can be made of MOGL to improve its effectiveness, and a full implementation is still necessary before it can go into more general usage. However, the results from this study have been encouraging enough to make this type of future work appear reasonable. Perhaps other similar microworlds could also be created for the exploration of other mathematical concepts.

This thesis involved graphics research, the concept of microworlds, human factors research, and some learning theory as part of the initial design. By taking these steps, and then using more traditional computer science techniques for the detailed design and implementation, it has been possible to build an effective computer subsystem for the exploration of motion geometry.

References

Abelson, Harold. Logo for the Apple II. Peterborough: BYTE/McGraw-Hill, 1982.

Abelson, Harold and Leigh Klotz, Jr. Logo for the Apple II Technical Manual. Cambridge: Terrapin, Inc., 1982.

Allan, Jeff. Geometry Toolbox (GT) Programmer's Manual. University of Calgary, 1987.

Barr, Avron and Edward A. Feigenbaum, eds. The Handbook of Artificial Intelligence. Vol. 1. Stanford: HeurisTech Press, 1981.

Brown, John Seely and Richard R. Burton. "Diagnostic Models for Procedural Bugs in Basic Mathematical Skills." Cognitive Science 2 (1978): 155-192.

Burton, R. R. "Diagnosing bugs in a simple procedural skill." Intelligent Tutoring Systems. Eds. Sleeman, D. and Brown, J. S. London: Academic Press, Inc., 1982. 157-183.

Card, Stuart K., William English, and Betty J. Burr. "Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys, and Text Keys for Text Selection on a CRT." Ergonomics 21, no. 8 (1978): 601-613.

Card, Stuart K., Thomas P. Moran, and Allen Newell. "The Keystroke-Level Model for User Performance Time with Interactive Systems." Communications of the ACM 23, no. 7 (1980): 396-410.

Cardelli, Luca and Rob Pike. "Squeak: a Language for Communicating with Mice." Siggraph '85 19, no. 3 (1985): 199-204.

Carroll, John M. and John C. Thomas. "Metaphor and the Cognitive Representation of Computing Systems." IEEE Tranactions on Systems, Man, and Cybernetics 12, no. 2 (1982): 107-116.

Cohen, Paul R. and Edward A. Feigenbaum. The Handbook of Artificial Intelligence. Vol. III. Stanford: HeurisTech Press, 1982.

Foley, James D. and Andries Van Dam. Fundamentals of Interactive Computer Graphics. Reading: Addison-Wesley, 1984.

Greenberg, Saul and Ian H. Witten. "Adaptive personalized interfaces - A question of viability." Behavior and Information Technology 4, no. 1 (1985): 31-45.

Greenberg, Saul and Brian Wyvill. A Tutorial Guide to Groper. University of Calgary Research Report, no. 83/131/20, 1983

Harrison, Bruce, Selwyn Brindley, and Marshall Bye. "Allowing for Student Cognitive Levels in the Teaching of Fractions and Ratios." Journal for Research in Mathematics Education 20, no. 3 (1989): 288-300.

Harrison, John. A Novel Computer Language Based on Transformational Geometry. Third Year Project. University of Warwick, 1985.

---. A User Interface for a Programming Language. Term Project for Computer Science 481. University of Calgary, 1986.

Harrison, Marilyn and Bruce Harrison. "A Math Lab for Every Student." The Alberta Teachers' Association 64, no. 1 (1983): 32-35.

Hill, D. Designing for Human-Computer Interaction: some rules and their derivation. University of Calgary Research Report, no. 84/166/24, 1984.

Hollnagel, Erik and David D. Woods. "Cognitive Systems Engineering: New wine in new bottles." International Journal of Man-Machine Studies 18 (1983): 583-600.

Kieras, David and P. G. Polson. "An approach to the formal analysis of user complexity." International Journal of Man-Machine Studies 21 (1985).

Lewin, Ann White. "Down with Green Lambs: Creating Quality Software for Children." Theory into Practice 22, no. 4 (1983): 272-280.

Miller, George A. Psychology of Communication. New York: Basic Books, Inc., 1967.

Newman, William M. and Robert F. Sproull. Principles of Interactive Computer Graphics. New York: McGraw Hill Book Company, 1979.

Paliès, Odile et. al. "Student modelling by a knowledge-based system." Computer Intelligence 2 (1986): 99-107.

Papert, Seymour. "Microworlds: Transforming Education." Artificial Intelligence and Education Volume One. Eds. Lawler, Robert W. and Yazdani, Masoud. Norwood: Ablex Publishing, 1987. 79-94.

---. Mindstorms. Sussex: The Harvester Press, 1980.

Park, Ok-choon and Robert D. Tennyson. "Computer-based Instructional Systems for Adaptive Education: A Review." Contemporary Education Review 2, no. 2 (1983): 121-135.

Piaget, Jean, and Barbel Inhelder. The Psychology of the Child. New York: Basic Books, Inc., 1969.

Rich, Elaine. "Users are individuals: individualizing user models." International Journal of Man-Machine Studies 18 (1983): 199-214.

Rochkind, Marc J. Advanced C Programming for Displays. Englewood Cliffs: Prentice Hall, 1988.

Skemp, Richard R. Intelligence, Learning and Action. Chichester: John Wiley & Sons, 1979.

---. The Psychology of Learning Mathematics (Expanded American Edition). Hillsdale: Lawrence Erlbaum Associates, 1987.

---. Structured Activities for Primary Mathematics. vol. 1. London: Routledge, 1989.

Thompson, Patrick W. "Mathematical Microworlds and Intelligent Computer-Assisted Instruction." Artificial Intelligence and Instruction. Ed. Kearsley, Greg P. Reading: Addison-Wesley Publishing Company, 1987. 83-107.

van den Bos, Jan. "Whither device independence in interactive graphics?" International Journal of Man-Machine Studies 18, (1983): 89-99.

Wyvill, Brian. An Interactive Graphics Language. Ph.D. diss., Bradford Univ., 1975.

APPENDIX A

The Survey Used in the Experiment

MOGL Survey

You will be given a number of tasks to complete using the computer language called MOGL, which allows you to create pictures on the computer screen, using motion geometry.

Try to complete all of the tasks indicated, and don't be afraid to ask for help. If you have problems with anything, write them down or mention them to the supervisor. Also write down any improvements you would like to see made to the system.

MOGL allows you to use shapes called "sprites." A sprite can be moved to different positions on the screen, and made to leave a "footprint" behind at any time. By putting footprints together, you can build up a picture.

This exercise uses menus to make selections rather than typing commands in. The menu is on the right side of the screen. To select items from the menu, move the cross hairs over the item to be selected (using the arrow keys on the keyboard), and then press the space bar. The item you select will either cause another menu to appear for further selections, or will cause a command to appear on the left side of the screen, as if you had typed it.

1. Call up the line sprite by selecting the **SPRITE...** menu, and then selecting **line** from that menu.

   Comments:

2. Make the sprite leave a footprint by selecting **mark**.

   Notice that the sprite is now hidden behind the grey footprint that it has left. You can think of this footprint as a mark left behind on the glass of the terminal screen.

   Comments:

3. Turn the sprite so we can see it again by selecting **MOVE...** to get the menu of various ways to move the sprite, and then **turn...** to turn the sprite. You can select the amount to turn from the dial that is now presented; select various positions around the circle until the line points to the number 45. Now, select **ACCEPT** to accept that value.

   Experiment with the turn command until the sprite is straight up and down. Notice that the sprite always turns around the centre of the screen.

   Comments:

4. Now, try sliding the sprite across the screen by selecting **MOVE...** and **slide....** You will be asked for a distance to slide the sprite. You will be given a scale; experiment with selecting different positions on the scale until you have selected the number **1**. Select **ACCEPT** to accept this entry. Now, you will be asked for a direction to slide the sprite (using a dial, as in step 4), so select a direction of **0**.

   This should slide the sprite about 1 centimeter to the right. Try the same thing again, except selecting a direction of **180**.

   Which direction does the sprite move this time?

   ____

   Experiment with sliding the sprite in different directions. Then, try changing the distance so that the sprite slides further than 1 centimeter (try not to slide the sprite off the edge of the screen!).

   Comments:

5. Select **clearscreen** to clear the screen. Select **line** to get the line sprite back in the original position. Now, using the commands turn, slide, and mark, try to draw a square made up of footprints on the screen. Write down the commands you use on the following sheet of paper (the commands will appear on the left side of the screen as you select them).

Comments:

6. MOGL allows you to make your own new sprites. For example, you might want to make a sprite which is a square rather than a line. This can be done using a program. A program is a sequence of steps you enter into the computer which can be executed later simply by typing the name of the program.

   To make a new sprite which is a square, select **LEARN....** You will be given a new screen, and MOGL will remember all the commands you type.

   You can now give the commands to create a square that you wrote down above, except that as you enter them, the computer memorizes them. Start by selecting **line**, and continue with your own commands. Every time you use the **mark** command, the footprint left behind becomes part of the sprite you're defining.

   When you have entered all the commands to make up a square, select **end** to tell MOGL your program is done. Now, the program screen will disappear, and you will be given the original screen back.

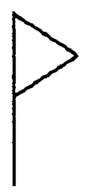
   You will be asked for a name for the new program, so type in **square** and press the RETURN key.

   You have now defined a new sprite called "square." Now whenever you select **square** from the **SPRITE...** menu, the sprite you have defined will appear on the screen. It can be manipulated in the same way as the line sprite.

   Select **SPRITE...** and then **square** to bring your sprite to the screen, and try various commands such as turn, slide, and mark on it.

   Comments:

7. You can also make more complex sprites. For example, say you wanted to make a flag that looks like the following:



This flag is made up of two parts; a triangle and a line. It could be made by defining a triangle sprite, and using that and a line sprite.

This has already been done for you; to load in this pre-defined program, *type* exactly the following:

load "flag

This defines two new sprites, called "triangle" and "flag." They will be available on the SPRITE menu.

Comments:

8. Call up the flag sprite using **SPRITE...** and **flag**.

   The flag sprite you have loaded can now be reflected in a mirror. To call up a mirror, select **MIRRORS...** then **mirror**. Select **1** from the scale. This calls up the first mirror (there are 8 mirrors available, numbered from 0 to 7). Notice that it is represented as a dashed line.

   Now, reflect it in this mirror by selecting **MOVE...** and then **reflect**. Select **1** from the scale to reflect it in mirror number 1.

   You can move the mirror into a different place using MIRRORS... and then the selections mturn and mslide. These work just like turn and slide, except that you have to tell which mirror to use. For example, try turning the first mirror 45 degrees. See what happens when you reflect the flag in this mirror. Now, using just the reflect and mturn commands, try to return the flag to its original position.

   Comments:

9. Now, use the commands you have learned to draw a picture of some sort. Any new shapes you need can be defined using programs, and you can put all the shapes together to make anything you can imagine.

   Comments:

Rating of MOGL

Rate MOGL in each of the following areas. Circle a number from 1 to 5, where 1 is poor and 5 is excellent:

1. The idea of sprites 1 2 3 4 5

   Comments:

2. Doing rotations of shapes 1 2 3 4 5

   Comments:

3. Doing slides of shapes 1 2 3 4 5

   Comments:

4. Changing sizes of shapes 1 2 3 4 5

Comments:

5. Doing reflections of shapes 1 2 3 4 5

Comments:

6. Making new sprites 1 2 3 4 5

Comments:

7. Drawing pictures using MOGL 1 2 3 4 5

Comments:

8. The menus 1 2 3 4 5

Comments:

9. Overall ease of use 1 2 3 4 5

Comments:

Comments

Write down any comments you have on this page. Anything you find interesting about MOGL, or things that caused you frustration are the types of things to write here. How would you improve MOGL?