C. 1

MODELLING AND ANIMATING THREE-DIMENSIONAL ARTICULATE FIGURES

by

DANNY G. CACHOLA

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Computer Science

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1986

78

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date April 25 / 1986

ABSTRACT

This thesis describes an animation extension to a high-level graphical programming language which provides constructs for the definition, manipulation, and external representation of three-dimensional articulate figures and and their associated movements. The extension permits the definition of models consisting of segments and joints and the specification of each model's motion at a high level of abstraction. The relationship of the extension with respect to the host language is discussed and a general description of the animation language's design and implementation is given. The modelling and motion constructs are discussed and examples of the constructs are presented. It is concluded that high level animation permits the implementation of sophisticated application programs that are easy to read and understand.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

"Animation is the graphic art which occurs in time. Whereas a static image (such as a Picasso or a complex graph) may convey complex information through a single picture, animation conveys equivalently complex information through a sequence of images seen in time. It is characteristic of this medium opposed to static imagery, that the actual graphical information at any given instant is relatively slight. The source of information for the viewer of animation is implicit in picture change: change in relative position, shape, and dynamics. Therefore, a computer is ideally suited to making animation 'possible' through the fluid refinement of these changes." Baecker [4]

Mankind has always been interested in pictorially representing his world. This is understandable; pictures can invoke emotions, refresh memories, teach, and explain. They allow the representation of a maximum amount of information in a minimum area.

Within the last one hundred years, a new interest has developed; the desire to add movement to pictures. The use of animation has been widely accepted and moving pictures have become common, however, the creation of animation is a labour-intensive process. For each second of viewing time, high quality-film animation requires the preparation of twenty-four images (frames), each differing only slightly from the preceding one. A two-hour animated feature film contains approximately 170,000 frames.

In an attempt to decrease the labour involved, commercial studios have traditionally painted images onto celluloid sheets called "cels". Several of these overlays are used to create a frame. This technique saves time, because only the parts of a character that actually move in a particular frame have to be redrawn. The entire process can still be quite expensive, since a feature film may require the preparation of over half a million cels [6].

With the introduction of computers, a faster and cheaper method of producing animated films was made possible. The first efforts to utilize digital computers in the production of animated films dates back to the early 1960s. Computer-assisted animation has since played a role in the three main areas of conventional animation: the creation of drawings, the production of in-between frames, and the painting of cels. Recent advances in computer graphics hardware and software have made computer animation a rapidly expanding field which now includes a large number of different production styles, approaches, and techniques.

One new approach that has been developed is computer-modelled animation. Computer-modelled animation differs from computer-assisted animation because it corresponds to animation sequences in which three-dimensional models move about in a three-dimensional space. This process is very complex without a computer. Many of the advances which have occurred in computer modelling in the last few years have been in the area of figure modelling and motion specification.

Several methods have been proposed including the modelling and control of figures using procedures (procedural modelling) [13], the control of a physical model by the application of forces (dynamic modelling) [1], the use of goal-directed systems for the generation of a model's motion [12, 23], and the use of key frame animation [7], one of the oldest animation techniques still in use.

A different approach for the modelling of a three-dimensional articulate figure and the subsequent control of its motions will be presented here. Whereas many of the previous methods have approached figure modelling from a subroutine level, this approach involves the use of a high-level animation programming language. It provides constructs that permit a programmer to define and manipulate data of the type MODEL and MOTION, therefore allowing the representation of real three-dimensional jointed bodies and their associated motions [24].

This thesis discusses aspects of the design and implementation of such a language. The problem of figure modelling and motion specification is dealt with in terms of kinematics: the study of position (displacement) and its time derivatives (velocity and acceleration). Considerations of force, mass (dynamics) [11, 17], balance [15], and obstacle avoidance [18] are beyond the scope of this thesis.

Chapter 2 presents an overview of the animation language. The overall relationship of the language with respect to the host language is considered. In addition, a general description of the language's design and implementation is given.

The language's modelling constructs are studied in Chapter 3. The internal representation of a figure and its implementation is discussed. Also, two different techniques for modifying a defined model are presented.

Chapter 4 introduces the concept of a motion primitive and examines the constructs which allow the motion's definition and manipulation. The motion's internal representation is analyzed and its implementation is presented.

Chapter 5 examines how the articulate figures and motion primitives are integrated to produce animation. In addition, several language features are introduced which simplify the animation process.

Chapter 2

LANGUAGE DESIGN ENVIRONMENT

Virtually all graphics programming languages are extensions to existing high-level computer languages, since a graphics application program generally requires non-graphics constructs for support. The same applies to the animation language discussed here. It assumes the availability of a high-level graphics programming language which provides the support for the non-animation constructs present in an animation application program. The constructs and features introduced are independent of any of the host constructs, consequently the animation extension can be applied to any high-level graphics language with equal success.

The host language used in the implementation is called LIG6 (Language for Interactive Graphics Version 6) [19, 22] and is currently in use only at the University of British Columbia. The LIG6 system is implemented on a 48 megabyte Amdahl 5850 mainframe, under the Michigan Terminal System. The language LIG6 was chosen for two main reasons: it is an easy language to learn, and it has been designed as a production system. Also, a large number of students have been exposed to LIG6, thus ensuring that the language is free of errors. Since LIG6 is the host language, the animation language implementation is called LIG ANIMATE (Language for Interactive Graphics ANIMATion Extension).

LIG6 is implemented as a FORTAN extension. A preprocessor written in PASCAL converts LIG6 programs into standard FORTRAN programs with extension elements translated into calls to subroutines in a run-time library. The subroutines are coded in FORTRAN. When a LIG6 application program is to be executed, the object deck produced by compiling the preprocessor output is run in conjunction with the run-time library; more complete information is available in the LIG6 User's Manual [20].

Since the LIG ANIMATE constructs are independent of its host language, the translator for the language is implemented as an independent preprocessor. The preprocessor analyzes only the animation constructs present in the animation application program and produces LIG6 target code. The decision to use a preprocessor has proven to be satisfactory. It has allowed experimentation with different animation constructs and has allowed experimentation with the preprocessor itself. In addition, the construction of a complete compiler for LIG6 and the extension would have complicated the objective of this thesis.

The LIG ANIMATE preprocessor was written with the use of a top-down Compiler Writing System (CWS). A CWS aids in the creation of a compiler or precompiler (preprocessor) for a general language, hence, it was unnecessary to completely write the preprocessor. Several consequences result from the use of a CWS in creating the preprocessor. The CWS does not support the free form input conventions of LIG6; delimiters and reserved words are required [16]. Consequently, the host language statements can not be parsed; it is necessary to flag the extension statements with a special character (an asterisk) in the first column.

The segregation of the host language and extension statements introduces two restrictions. The preprocessor does not allow the mixture of host and extension statements on a single line. Any syntactic errors which are present in the host statements are not detected until the preprocessor output is compiled. The objective of the research was not the creation of a complete production system. Its object was to examine the definition of animation at a high level of abstraction, therefore, the approach was chosen to simplify the implementation.

LIG ANIMATE has been designed independently of the host language, hence, it does not take advantage of many of the host's construct features. For example, LIG6 allows the use of COMMON statements, arrays, functions, and parameter passing of the basic data types. LIG ANIMATE does not allow these features with the use of its data type MODEL, although, MOTION data types can be used in COMMON statements and passed as parameters.

The implementation does share some general input conventions with LIG6. Although the preprocessor introduces reserved words which must be delimited by blanks, statements may span two or more lines. Multiple extension statements per line are allowed, provided they are separated by semicolons. Comments enclosed in braces may appear anywhere. The length of a line is 255 characters and column positions are not important with respect to the beginning of statements.

Several LIG6 input conventions have been extended in LIG ANIMATE. Variable names, present in the animation constructs, may contain up to fifteen characters, however, the first five characters must be unique. Identifiers, variables, and keywords used in LIG ANIMATE may be in either upper case or lower case, since they are translated to upper case. The only consideration which affects any of the host code statements is that none of the variables present in the host statements or animation constructs can begin with the dollar sign. Variables beginning with the dollar sign are restricted for use by the LIG6 ANIMATE routines.

Two factors helped further shape the eventual design of the preprocessor. The CWS is based on PASCAL and LIG6 is based on FORTRAN. A problem is created since the manipulation of articulate figures is naturally recursive. FORTRAN does not easily allow the simulation of recursion, therefore, the model database is stored in the preprocessor. The preprocessor receives commands to define, manipulate, and display models in its database. When the preprocessor displays the model, it does so by creating program segments that LIG6 can process.

The motion database is not stored in the preprocessor. A suitable representation was implementable in the host language, consequently, motions are stored in the host language. The preprocessor directly translates motion constructs into a form which the LIG6 language can compile. The motion extensions are translated into calls to subroutines in a run-time library. The subroutines are written in LIG6. When a LIG ANIMATE program is to be executed, the object deck produced by compiling the LIG6 output is run in conjunction with the run-time library.

Chapter 3

MODELLING SYSTEM

Before an attempt is made to specify a desired motion for a figure, a method for specifying the figure must be available. This chapter discusses the problems associated with model specification and presents two constructs that allow the creation of an articulate model.

LIG ANIMATE assumes that all of the model's segments (links) are rigid, that is, once defined, the links are assumed to remain the same shape throughout the life of the model. No formal specification is incorporated in LIG ANIMATE for the definition of links, they are assumed to be defined as graphical objects in the host language. Every segment is assumed to have been defined in its own local coordinate system [3, 12, 23], with the origin at the center of the link.

3.1 DESCRIPTION OF JOINTS

In the simplest form, an articulate model contains two rigid segments and a joint between the segments. Since the responsibility for defining, manipulating, and displaying the links is that of the host language, LIG ANIMATE is only responsible for the specification of the relationship between the links. The specification includes the position, degrees of freedom, and restrictions on a joint.

A joint has up to three degrees of freedom; it can be rotated about the X, Y, and Z axes. Joints may be restricted to fewer than three degrees of freedom by permitting the joint to rotate about only one or two of the axes. Thus, simple joints such as fingers (hinge joints), and complex joints, such as shoulders (ball-and-socket joints) can be simulated. A joint connects only two segments. A joint can move independently of all other joints, hence, the position of one joint does not affect the motion of another.

Links are restricted in their movements about a joint. During a single joint's movement, one link (the primary segment) is considered stationary and the second link (the secondary segment) moves with respect to the stationary link. A single link can function as both a primary segment and a secondary segment if it belongs to more than one joint. One segment, the model's main link, is singled out from the others. All movement ultimately refers to the main link. Only one segment may be designated as the main link and it must be the primary segment in all joints it belongs to.

Each joint instance is assigned a unique identifier that permits subsequent reference to the joint. The user may place restrictions on the range of angles through which the secondary link can travel and may specify where the two segments are to be joined. A typical statement creating an articulate joint is

```
JOINT joint_identifier,
      primary_link,   relative_location_1,
      secondary_link, relative_location_2,
      x_extremes, y_extremes, z_extremes
```

where 'joint_identifier' is the unique identifier for this joint instance; 'primary_link' and 'secondary_link' are previously declared graphical objects (which are defined in independent coordinate systems). 'primary_link' is the stationary segment with which 'secondary_link' moves. 'relative_location_1' and 'relative_location_2' are vectors that contain the joint's relative positions, with respect to each graphical object's coordinate system. The extreme parameters are component vectors which store a pair of maximum angles beyond which the joint can not move. Extremes, as well as the joint's current angles, are specified relative to the joint's predefined neutral position of (0°, 0°, 0°).

The neutral position for a joint is determined by the definition of the graphical objects used in the joint. Two different joints are shown in Figure 1. In both cases, the models are present in their neutral positions which is possible only if the segments are defined as in Figure 2 and the joint locations are specified as in the subsequent statements.

```
     JOINT joint_id, object1, ( 0.0, -2.0, 0.0),
                     object2, ( 0.0, +2.0, 0.0),
                     x_extremes, y_extremes, z_extremes

     JOINT joint_id, object3, ( 0.0, -2.0, 0.0),
                     object4, (-2.0,  0.0, 0.0),
                     x_extremes, y_extremes, z_extremes
```

Both images in Figure 1 can be produced by either of the joints presented, if the current angles in either joint are +90°, or -90.0°, respectively.

The direction of the extremes associated with a joint can be determined by the right hand rule. By pointing the right hand thumb in the direction of the positive half of an axis and closing the right hand, the counterclockwise direction of the curled fingers is the positive direction for the extremes. Consequently, the clockwise direction about the axis is the negative direction for the extremes. Figure 3 displays a joint in its neutral position which contains one degree of freedom since the X and Y axes are locked at 0°. The joint's axes and the associated maximums are also displayed in the figure. The joint can be created by the statement:

```
     JOINT joint_id, object1, rel_loc1,
                     object2, rel_loc2,
                     (0°, 0°), (0°, 0°), (0°, 135°)
```

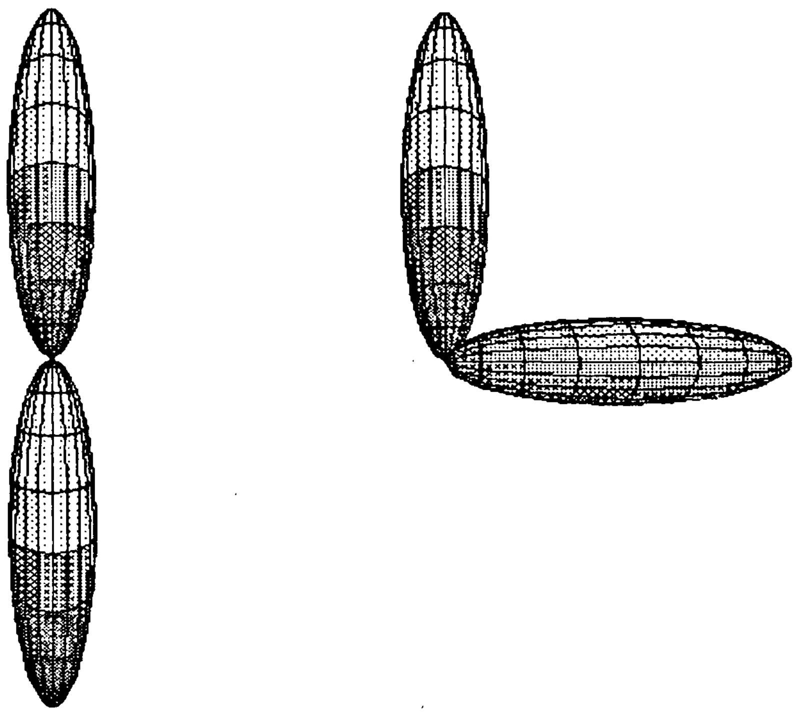

Figure 1

Two joints in different neutral positions

3.2 INTERNAL REPRESENTATION

The internal representation of an articulate figure is described by a tree structure of nodes and arcs that is stored in the LIG ANIMATE preprocessor. Links are represented by nodes and the joints are represented by arcs [2, 23]. Each level of nodes moves with respect to the nodes of the higher level and is considered stationary by the nodes below. The nodes at the leaves of the tree represent the outermost extremities of the model; the root node is considered the main link. Figure 4 is an example of

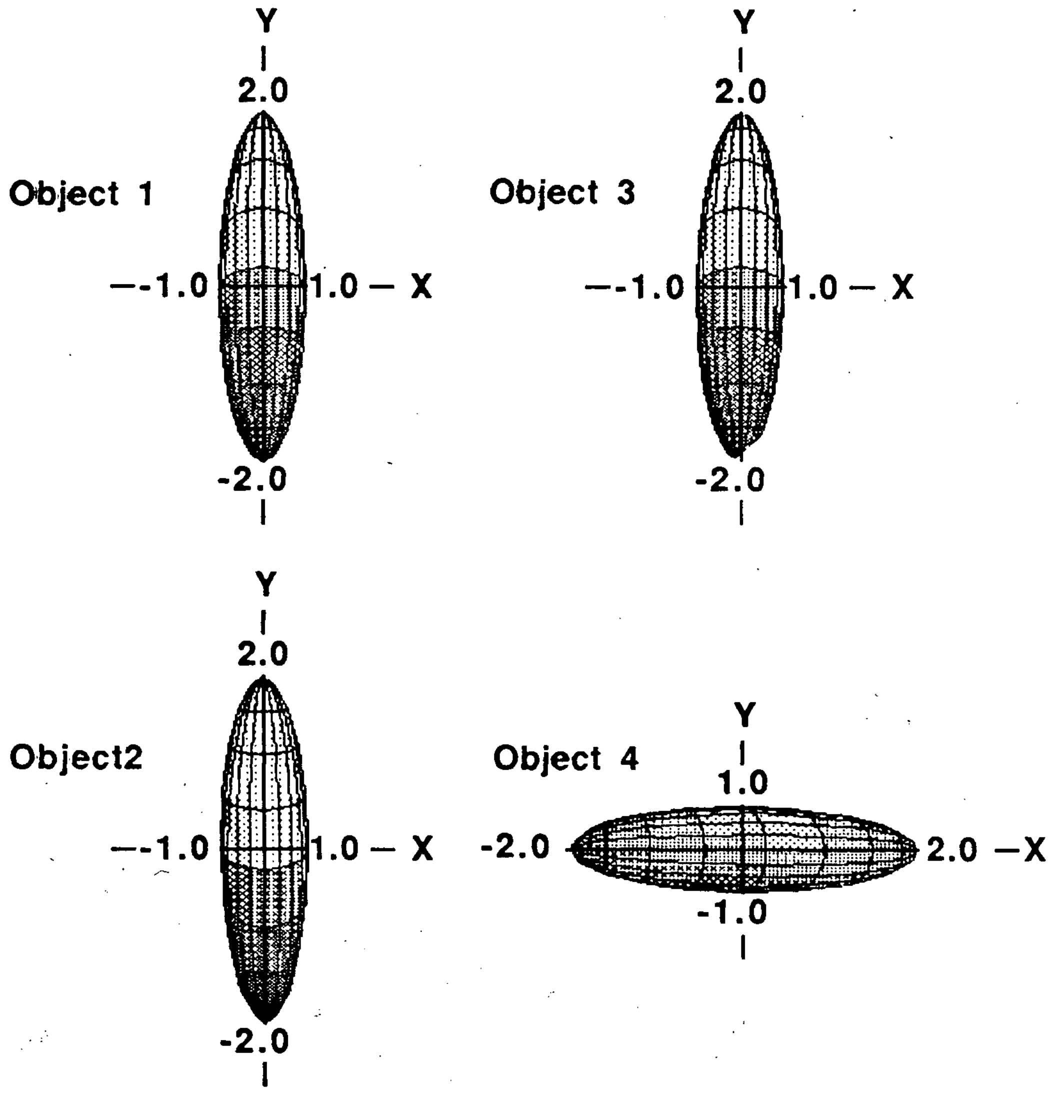

Figure 2

Definition of graphical objects

a partial model of a human figure. Figure 5 contains the model's internal structure. A model may have at most a total of m arcs present in its structure. This value is theoretically unlimited, however, LIG ANIMATE currently restricts models to a maximum of thirty arcs (joints).

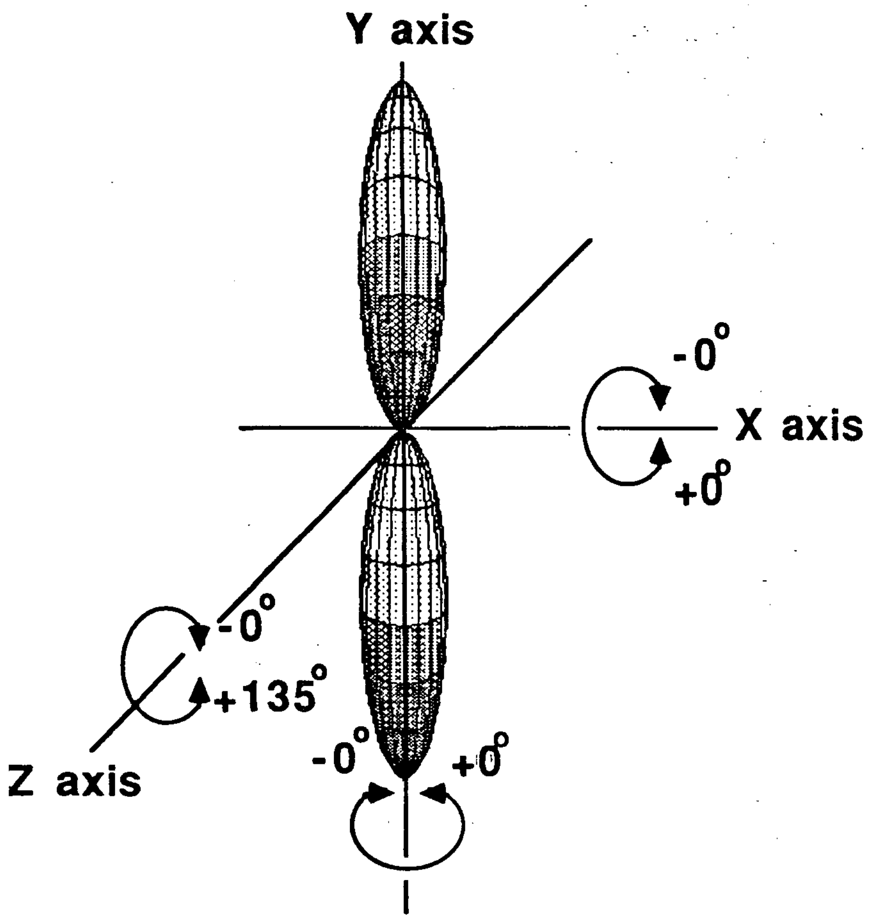

Figure 3

Sample joint extremes

Each node is associated with at least one arc, as in the case of the extremities that are secondary segments. Typically, two arcs are associated with a node, corresponding to a link (such as the femur of a leg) that acts as both a primary and secondary segment, however, a node may have three or more arcs associated with it. For example, the hand has six arcs (one representing the wrist joint, the other five representing the finger joints). A

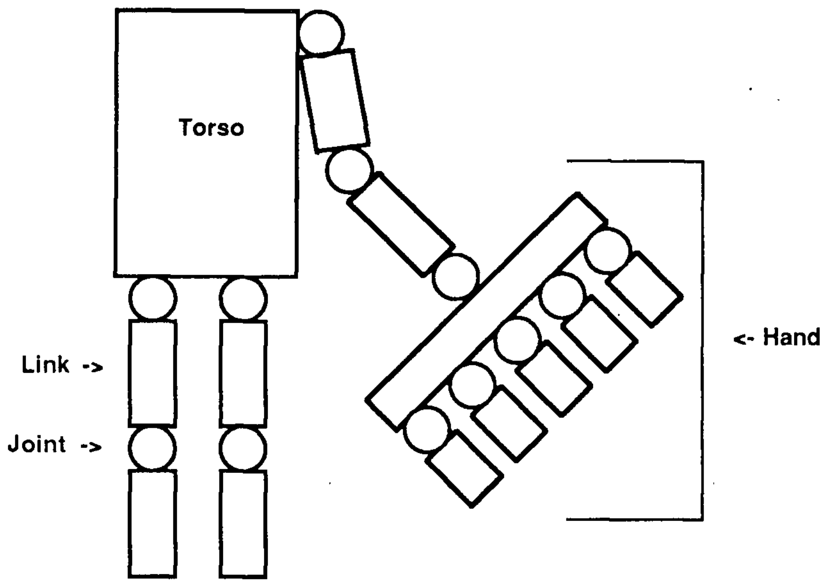

Figure 4

Partial model of a human figure

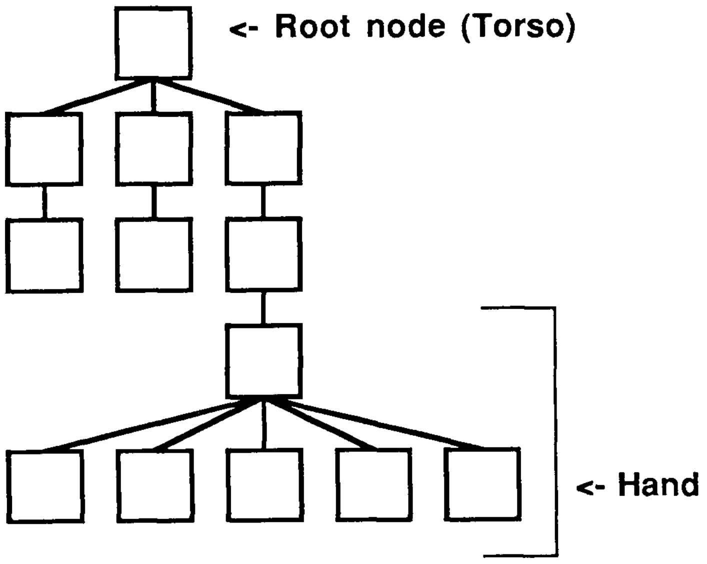

Figure 5

Internal structure of a model

node may be associated with at most n arcs. The branching factor n is also theoretically unlimited, but a factor of ten is deemed sufficient to define most articulate figures.

The information typically associated with a node is shown in Figure 6. Every node contains general information such as the name of the corresponding host's graphical object, the scaling factor, and the number of joints present in which the current node is a primary link. In addition, every node carries joint specific information. Both of the nodes bounding an arc contain information pertaining to the joint and a pointer to the joint's opposite node. Each node also contains a joint identifier and the location of the joint on the node's graphical object. The joint identifier is used to identify a joint and acts as a pointer to information stored at the model level containing the restrictions associated with the joint. The joint information stored at table location zero represents a joint in which the current node is a secondary segment (an upward arc). The joint information stored in locations one to n represent joints in which the current node is a primary segment (a downward arc).

The model's tree structure permits the representation of open kinematic chains only. A kinematic chain is a linear sequence of links that are connected by joints. In an open chain, one end point is fixed and the remaining

Graphical Object Name			
Object Scale			
Number of Subjoints			
No.	Joint #	Ptr to Node	Joint Location
0	0	0	(0,0,0)
1	0	0	(0,0,0)
2	0	0	(0,0,0)
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
n	0	0	(0,0,0)

Figure 6

Information associated with each link

chain is allowed to move freely, as in Figure 7a. In a closed chain, more than one end point is fixed in space, as in Figure 7b. For example, if two hands are joined together and the arms are allowed to move while keeping the body motionless, a closed kinematic chain is formed by the arms. The motion produced in such a chain is more complex to analyze and is beyond the scope of this thesis.

Every model has an associated symbol table, see Figure 8. The table contains information from the declaration of the model, in addition to state information. The information includes: the joint identifiers, the rotational maximums associated with each degree of freedom, the current rotational angles of each degree of freedom, and the

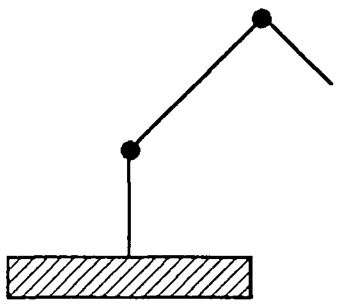
Figure 7a

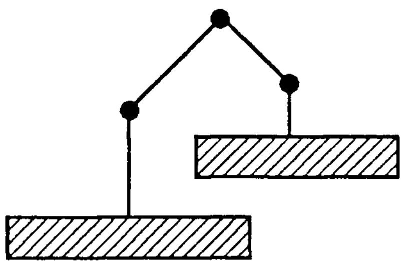
Figure 7b

a) Open kinematic chain
b) Closed kinematic chain

instantaneous velocity and acceleration of the joints along each degree of freedom. The table is defined to describe the model's position completely. The contents of the table, at a specific frame, represents a model's current orientation.

Joint #	Current Orientation	X Extremes	Y Extremes	Z Extremes	Current Velocity	Current Acceleration
No. 1	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	0	0
No. 2	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	0	0
No. 3	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮
No. m	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	0	0

Figure 8

Model symbol table

Since the state information must be available to the model during the animation, the table can not be stored in the preprocessor. The table is stored in a suitable form in the host language. Whenever the preprocessor receives a command to retrieve a model, the preprocessor transfers the model's external representation and the model's symbol table to the target code.

3.3 MODEL DEFINITION USING SUBMODELS

An articulate model can be defined using two basic techniques. The first uses the JOINT construct and allows the use of articulate submodels; the second approach is discussed in Section 3.4. Instead of simply using a static graphical object for every link, each link can consist of a grouping of other segments and joints, which permits the creation of intermediate models or submodels that can be referenced independently.

In the statements,

```
a :- JOINT .., obj1, .., obj2, ..
b :- JOINT .., obj3, .., a,    ..
c :- JOINT .., b,    .., obj4, ..
d :- JOINT .., a,    .., b,    ..
```

the symbols 'a', 'b', 'c', 'd' represent variables of the type MODEL. The symbol ':-' is the model assignment operator, which causes the resulting node structure on the right hand side of the MODEL variable to be stored on the left hand side of the operator. If each of the preceding statements were executed in order, the structures in Figure 9 would be generated respectively.

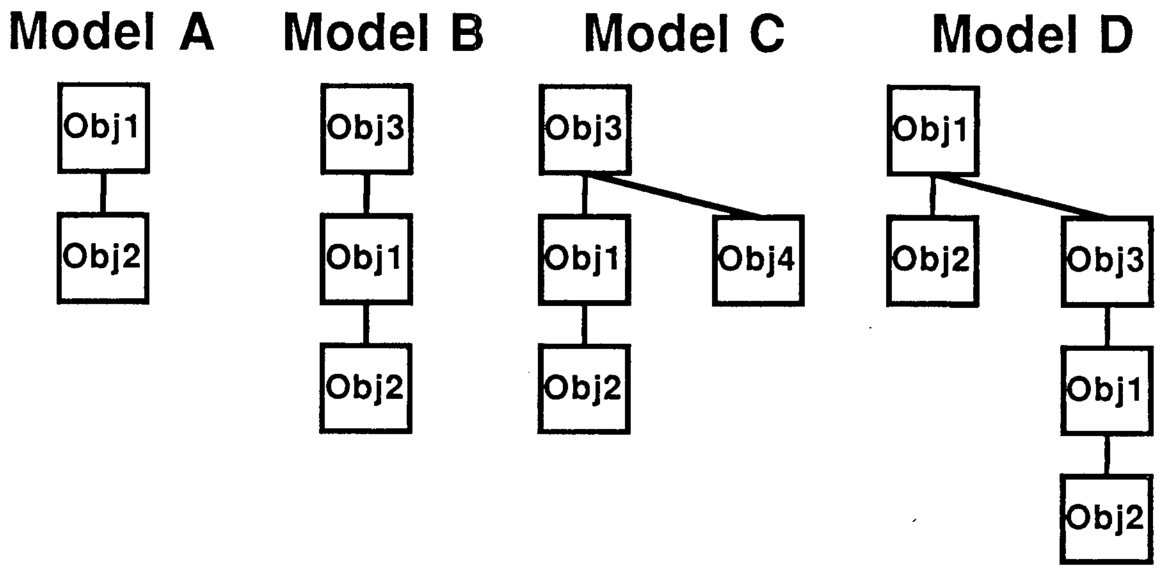

Figure 9

Model assignment

The JOINT construct can be compared to the '+' (superposition) in high-level graphical languages. Whenever the secondary segment is a model, the relative location vector for the secondary segment must be specified with respect to the main link in the submodel. Models are internally represented as pointer structures, hence, the LIG ANIMATE system does not allow the user to create cycles in the structures. Infinite loops are prevented since the system uses copies of the submodels specified in the JOINT construct, if the submodel's name is not the same as the recipient model. This approach is effective because it prevents the creation of infinite loops, such as by the statements:

```
     a :- JOINT .., b, .., c, ..
     c :- JOINT .., a, .., d, ..
```

The technique allows an efficient implementation of model addition. In the statements

```
     a :- JOINT .., a, .., b, ..
     a :- JOINT .., b, .., a, ..
```

model 'b' is added to model 'a', without using copies of both models. The system has one restriction. The following statement

```
     a :- JOINT .., a, .., a, ..
```

is not allowed. The execution of such a statement creates a structure containing a cycle, therefore, the preprocessor produces an error message.

An advantage to the above method of model creation is that fewer statements are required to create symmetric models. For example, creating a model of a human body first entails the creation of submodels for the right arm and the right leg. Once defined, both submodels can be duplicated and joined to the right and left half of a human torso. A separate set of left limbs need not be defined. One problem arises. The values of the joint identifiers should not be duplicated in the right hand and left hand limbs, else the two sets of limbs will behave identically. Thus, to ensure unique identification of the joints, the submodel joint identifiers must be modified before the submodels can be connected to the torso.

Consider the statement:

```
arm := JOINT 2, humerus, (1.0, 5.0, 1.0),
                forearm <TRANS JOINTS BY 10>,
                (0.0, 0.5, 1.0),
                (0°, 135°), (0°, 180°)
```

which creates a model of an arm consisting of a forearm and backarm (humerus) joined at the elbow. The secondary link (forearm), which was previously defined, is an articulate model whose joint identifiers have been translated by a value of ten onto a new range of identifiers.

3.4 MODEL DEFINITION BY CONSTRUCT

The submodel approach has one disadvantage. It can produce temporary models unnecessarily. Consequently, an alternative technique for creating models is provided that assumes none of the model's subcomponents is subsequently required. The method creates only the resultant model with no intermediate articulate models. The construct

```
STARTMODEL arm
    JOINT 1, humerus, (0.0, 5.0, 1.0),
             ulna, (0.0, 0.5, 1.0),
             (0°, 35°), (0°, 180°)
    JOINT 2, ulna, (0.0, -5.0, 1.0),
             palm, (0.0, 3.0, 1.0),
             (-90°, 0°), (-25°, 25°)
    JOINT 3, palm, (0.0, -3.0, 1.0),
             little, (0.0, 1.5, 0.5),
             (-90°, 45°), (-45°, 45°)
    JOINT 4, palm, (0.0, -3.0, 2.0),
             ring, (0.0, 1.5, 0.5),
             (-90°, 45°), (-45°, 45°)
    JOINT 5, palm, (0.0, -3.0, 3.0),
             middle, (0.0, 1.5, 0.5),
             (-90°, 45°), (-45°, 45°)
    JOINT 6, palm, (0.0, -3.0, 4.0),
             index, (0.0, 1.5, 0.5),
             (-90°, 45°), (-45°, 45°)
    JOINT 7, palm, (0.0, -3.0, 5.0),
             thumb, (0.0, 1.5, 0.5),
             (-90°, 45°), (-45°, 45°)
ENDMODEL
```

creates a non-trivial model 'arm' with a single construct. The primary links specified within the construct are restricted to graphical objects, however, the secondary links may be either graphical objects or submodels.

If the model 'hand' comprising the last five joints has been defined prior to the use of the MODEL construct, then the model definition can be further simplified:

```
STARTMODEL arm
    JOINT 1, humerus, (0.0, 5.0, 1.0),
             ulna, (0.0, 0.5, 1.0),
             (0°, 35°), (0°, 180°)
    JOINT 2, ulna, (0.0, -5.0, 1.0),
             hand, (0.0, 3.0, 1.0),
             (-90°, 0°), (-25°, 25°)
ENDMODEL
```

The order of the JOINT statements present within the MODEL construct is arbitrary. The joint statements need not be specified in the same order as their appearance in the internal structure. The system sorts the specifications and assembles the model in the appropriate order, provided the segment identifiers are unique. If the segment identifiers are not unique, the JOINT statements must be placed in the same order as they appear in the internal structure.

The user is responsible for creating a non-ambiguous model specification. All the preceding model definitions produce models unambiguously. The following is an example of an ambiguous model:

```
STARTMODEL hand
    JOINT 1, palm,   ..,
             finger, ..,
             .., .., ..
    JOINT 2, palm,   ..,
             finger, ..,
             .., .., ..
    JOINT 3, finger, ..,
             fingernail, ..,
             .., .., ..
ENDMODEL
```

The specification of JOINT 3 causes the difficulty. With this model definition, the system cannot determine to which finger the fingernail belongs. The result of such a specification depends completely on the implementation of the preprocessor, hence, it should be assumed that ambiguous specifications produce unpredictable models.

3.5 MODEL TRANSFORMATIONS

Once a model has been defined, it can be treated as a unit. As previously shown, a model can function as a building block for the definition of more complex models. Currently, no constructs are provided that permit the user direct access to the model's database, however, several transformations have been provided that permit model modification by the user.

Model elements on the right hand side of a model assignment statement represent instances of previously defined model identifiers. Such instances can be modified creating different instances of a model identifier with different results. Currently, the modifications permitted are the transformations joint identifier translation, model segment scaling, and model joint extreme scaling. They follow a model in a list separated by commas and surrounded by angle brackets ("<", ">") and have the syntactic form

```
TRANS JOINTS [ BY ] <integer>
SCALE MODEL <real>
SCALE EXTREMES <real>
```

where brackets ("[" and "]") indicate an option and "<type>" represent constants of the type "type". Model transformations are independent, consequently, they are commutative. They may be applied in any order and may be repeated. If the TRANS JOINTS transformation is repeated, the resulting transformation is the sum of all the instances. Repeating either the SCALE MODEL or SCALE EXTREMES transformations produces a transformation consisting of the product of the instances.

The TRANS JOINTS transformation translates the joint identifiers present in the model. The statement

```
b :- a <TRANS JOINTS BY 10>
```

creates a new model whose joint identifiers have been translated by a value of ten onto a new range of identifiers. This transformation is used primarily whenever submodels present in a model definition contain an overlap in the joint identifier range.

The SCALE MODEL transformation scales all of a model's segments. The statement

```
a :- a <SCALE MODEL 2.0>
```

redefines model A such that its physical size is twice its original size. If the scaling factor is negative, the system will create a new model instance which is a mirror image of the previous model.

The SCALE EXTREMES transformation scales all of a model's joint extremes. The statement

```
b :- a <SCALE EXTREMES 2.0>
```

creates a new model whose joint extremes have been multiplied by the factor 2.0, thus doubling the range through which the joints can travel. If the scaling factor is negative, mirror images of the existing joint ranges are created, in addition to the change in the joint movement's magnitudes.

The statement

```
b :- a <SCALE MODEL 1.5, SCALE EXTREME 0.7>
```

creates a model 'b' which is larger than model 'a', and whose movements are more restricted than those of model 'a'. Model transformations are useful in creating models whose size and movement ranges are different, but whose internal structure are the same. If the preceding statement was executed, with 'a' predefined as a model of a robot, the corresponding models in Figure 10a and 10b would be generated. Figure 10a and 10b represent models 'a' and 'b', respectively, and the extremes by which they can squat.

Often it is desirable to have a model perform a range of motions. If these motions involve the movement of a model with respect to different main links, a technique must be available to change the main link with which a movement is performed. Therefore, a second method exists for the transformation of a model. The transformation has the

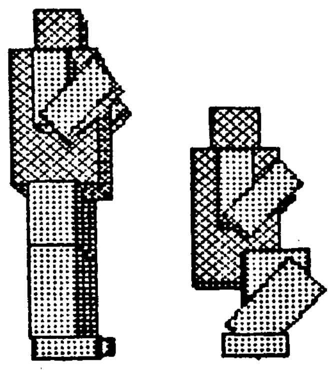
Figure 10a

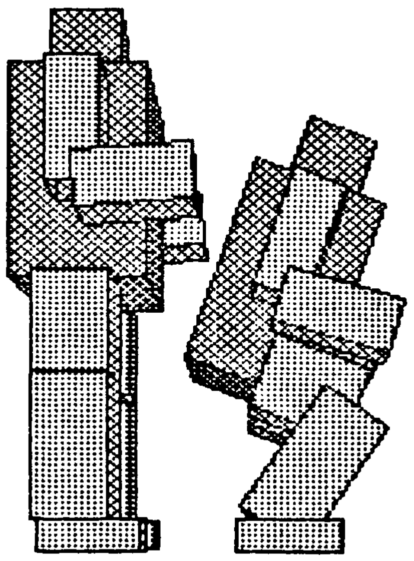
Figure 10b

a) Model squatting
b) Modified model squatting

syntactic form:

```
     FIGURE model_name MAIN [ LINK link_id ]
                            [ JOINT joint_id ]
```

Execution of the FIGURE statement assigns a new link as the model's main link. The new main segment may be specified by the link's name or by the joint identifier in which the link is present. If the joint identifier is used, the joint's secondary link is defined as the main segment. Executing the FIGURE statement without any options assigns the model's original main link as the current main segment. Figures 11 and 12 demonstrate the difference the main link can make to a movement. In Figure 11, the model has squatted correctly. The torso has descended towards the ground because the main segment is a foot. In Figure 12, the main link is the torso and the squatting movement produced is quite different; the feet have risen away from the ground. This occurs since the torso is kept stationary during the squatting motion.

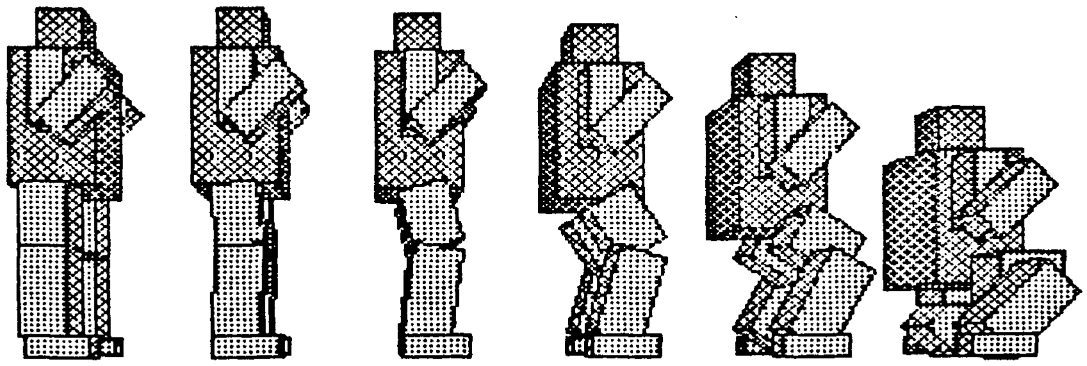

Figure 11

Model squatting with the foot as main link

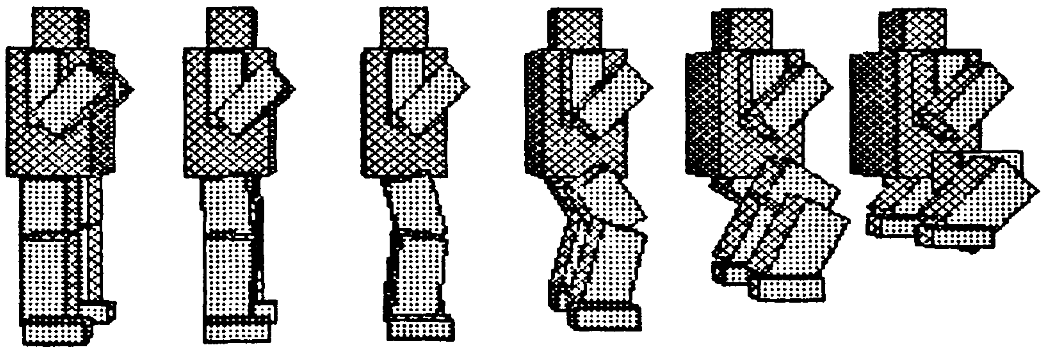

Figure 12

Model squatting with the torso as main link

The internal structures of the models are set up such that the structure is independent of the main segment. Yet, the structure enables the system to determine the original orientation of the structure. This ability allows the system to orient a model correctly whenever a new main link is defined.

A relationship exists between the sign of a joint's current angle and the priority of its two segments. In Figure 13, both joints have the same angle magnitude, however, the sign of the angles between the links is reversed. The sign of the angle depends upon which link is viewed as the primary or secondary link. Whenever a new main segment is defined, several of a model's links reverse their roles (i.e. primary links become secondary and secondary links become primary). By comparing the model's current main segment with the originally defined main link, the system can automatically determine which joint angles need their signs reversed.

In Figure 14, the model has squatted. The torso has descended towards the ground because the main segment is the left foot. The model was originally defined with the torso as the main link, consequently, the roles of the left foot and leg have changed. The signs of the joint angles, however, have not been reversed thus the left leg became contorted.

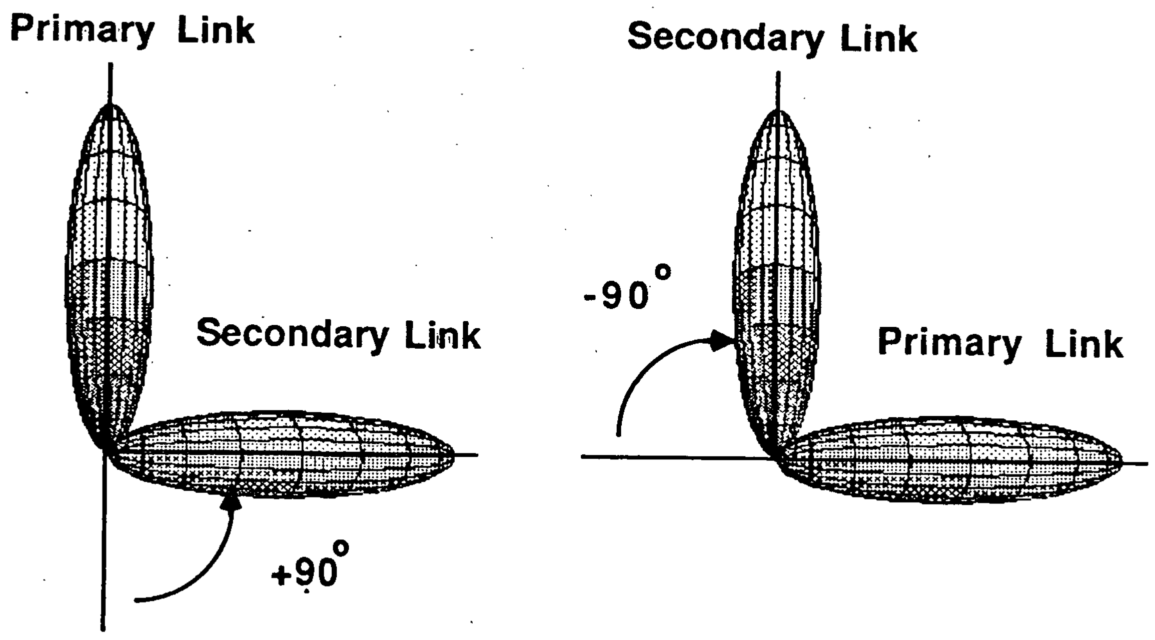

Figure 13

Joints with same angle but different sign

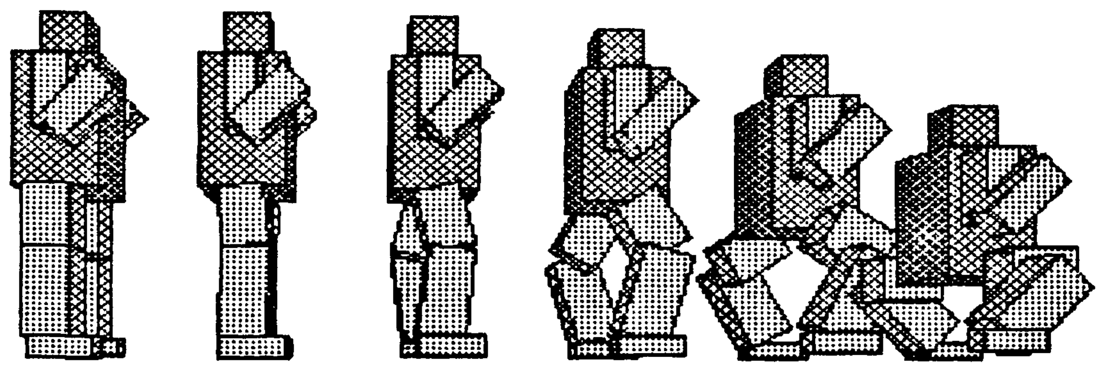

Figure 14

Model improperly squatting

Chapter 4

MOTION PRIMITIVES

Once the model has been defined, motions for the model can be created. This chapter discusses the problems associated with motion specification, presents two constructs for defining motions, and examines a method for motion manipulation.

Conventional two-dimensional animation relies heavily on the use of keyframe and interpolation ("in-betweening") techniques. Keyframes are a set of important drawings that show figures at crucial points (extremes) in a given action. In animation studios, keyframes are drawn by head animators; the missing frames required to create smooth animation are produced by in-betweening artists.

One of the earlier approaches to computer-assisted animation allowed the animator to enter keyframe drawings into the computer's memory and had the computer assume the role of the in-betweeners [7]. Although some very effective animations have been achieved, keyframe drawings are two-dimensional projections of three-dimensional figures visualized by the animator, hence the information available is limited in creating realistic animation. Also, in-between frames are frequently linearly interpolated, resulting in temporal discontinuities and movements that actually deform figures during the animation. For example, if a figure's leg obscures the other, the loss of information limits the automatic in-betweening of the keyframe drawings. An animator can deduce the original object from the drawings because he is familiar with the original model. In order for a program to understand a drawing it must contain a model of the figure that corresponds to the model in the animator's head [9].

Use of three-dimensional computer modelling in figure representation eliminates several problems associated with two-dimensional animation, however, the problem of motion specification for a three-dimensional model is introduced. One popular technique is to express a model's positions and velocities as functions of time. The technique is frequently used on a piecewise basis (i.e. the motion of each model segment is computed separately). Functions of time are evaluated on a frame-by-frame basis which involves specifying a path over time. The method has the advantage of producing motion with few temporal discontinuities, nevertheless, the description of a three-dimensional path as a function of time is generally a difficult task.

4.1 MOTION SPECIFICATION

The proposed approach for motion specification combines the best aspects of both the keyframe and functional techniques. Whereas two-dimensional keyframe animation views a figure with an external perspective and the functional approach views a figure from a piecewise perspective, the proposed technique treats a model as a unit and views it from an internal perspective (i.e. from the model's point of view). Consequently, a model's positional orientation can be uniquely specified throughout time. The method is similar to keyframe animation because keyframes or positional extremes are used throughout time, however, a three-dimensional model is employed and dealt with on a high level of abstraction, thus allowing more flexibility and accuracy in a figure's animation.

The motion specification of a single joint is similar to an animation script used in cel animation. As shown in Figure 15, the specification contains a joint identifier, an initial starting position (angle), and a set of movements (keyframes). Every keyframe contains the frame identifier during which the movement is completed, the joint position at the current frame, and the interpolation method used to reach the positional extreme (Figure 16).

Joint #	Starting Position	Movement 1	Movement 2	. . . .

Figure 15

Motion specification for a single joint

Frame #	Angular Position	Interpolation Method

Figure 16

Keyframe information

The angular position is given relative to the joint's neutral position of (0°, 0°, 0°). The neutral position is determined by the definition of the graphical objects used in the joint. The frame identifiers represent the number of ticks (units of time) that have passed during the motion sequence. The frame identifiers are given with respect to the start of the motion specification. The initial frame identifier is assigned the value of zero. The interpolation techniques currently available are: linear, acceleration, deceleration, and a combination of both acceleration and deceleration.

The keyframe information is sufficient to specify a single joint's movement throughout an animation sequence. The interpolation routines have been designed to reduce the effects of temporal discontinuities. Such reductions are possible because the system stores every joint's instantaneous velocity and acceleration. The interpolation routines take advantage of the information and make interpolation adjustments. For example, the linear interpolation routine allows several frames for a joint to achieve an optimum velocity before the joint travels linearly. Similar modifications have been made to all the interpolation routines, therefore decreasing the irregular motion which is normally caused by discontinuities in the direction of motion at every keyframe.

Joint 1	Starting Position	Movement 1	Movement 2	....
Joint 2	Starting Position	Movement 1	Movement 2	....
Joint 3	Starting Position	Movement 1	Movement 2	....
Joint 4	Starting Position	Movement 1	Movement 2	....
. . . .	. . . .	. . . .	. . . .	

Figure 17

Motion template

A complete motion definition for a model consists of motion specifications for all of the joints present in the model. A motion template is given in Figure 17. The motion specifications are entirely independent, consequently, different numbers of keyframes may exist for each joint and the keyframes may end on different frame identifiers. Any joint that finishes processing its keyframes remains frozen at its current angular position for the remainder of the motion.

4.2 EXPLICIT DEFINITION

A motion can be specified using two basic approaches, an explicit and implicit approach. The implicit approach is discussed in Section 4.3. The explicit approach allows an exact motion specification for a model, however, the details of every joint's orientation must be supplied to the LIG ANIMATE language.

The explicit specification of a model's motion can be approached with either of two techniques. The first technique assumes that the desired motion contains relationships between each joint's keyframes that can be expressed in the form of a mathematical formula. A typical statement defining one keyframe for a single joint is

```
motion_name [joint_id, key_frame_id] :=
   FRAME frame_id
       POSITION angle_x, angle_y, angle_z
       INTERPOLATE interpolation_technique
```

where 'motion_name' is a variable of the type MOTION; 'joint_id' is a joint identifier in the model for which the movement belongs. A motion may not have more joint specifications than the maximum allowable joints per model. Currently this number is thirty. 'key_frame_id' is the relative location of the keyframe from the beginning of the motion. A motion may have at most n keyframes. This value is theoretically unlimited, however, LIG ANIMATE currently restricts motions to a maximum of fifty keyframes.

'frame_id' is the frame location in the animation sequence at which the keyframe specifies the figure's movement extreme. 'frame_id is specified relative to the beginning of the motion. 'angle_x', 'angle_y', and 'angle_z' represent the angular position of the joint's secondary link with respect to the primary link. 'interpolation_technique' is the interpolation method (LINEAR, ACCELERATE, or ACCDCC) used from the previous keyframe to the current keyframe. If 'keyframe_id' is specified as zero then the value for FRAME is set to zero regardless of the value actually specified by 'frame_id'; the interpolation_technique is also ignored. The approach allows direct access to a motion variable. It can be employed within other constructs and permits the use of iteration to produce the motion specifications. The technique reduces the number of statements and the amount of work needed to define the specifications.

Since the construct allows the specification of a model's motion with the use of a formula, the identifiers may be either variables or constants. If variables are used, the variable value at the time of the statement execution will be stored in the current keyframe. The following is an example of a motion definition and a corresponding model definition.

```
DO 10 hand = 1, 2
    DO 20 minute = 0, 120
        IF hand = 1 THEN
        BEGIN
            { Calculate the minute hand's current }
            { angle                               }
            anglez = (360 / 60) * minute
        END
        ELSE BEGIN
            { Calculate the hour hand's current angle }
            anglez = (360 / 12) * (minute / 60)
        END

        anglez = anglez MOD 360
        angle = (0, 0, anglez)

        frame_num = 24 * minute

        TIME[hand, minute] = FRAME frame_num
                             POSITION angle
                             INTERPOLATION linear
20  CONTINUE
10 CONTINUE


clock :- JOINT 1, clock_body,  ...,
                  minute_hand, ...,
                  (0°, 0°), (0°, 0°), (0°, 360°)

clock :- JOINT 2, clock,      ...,
                  hour_hand, ...,
                  (0°, 0°), (0°, 0°), (0°, 360°)
```

The defined motion animates a clock for two minutes. To show an exaggerated movement of time, the clock runs at the rate of one minute per second. Therefore, in a two minute scene, two hours will have elapsed. Figure 18 shows six selected frames from the animation sequence. In the sequence, the clock's neutral position was defined at twelve o'clock. If a clock is to be animated over a different two hour range, this motion can animate a different clock defined with a neutral position at the start of the new range. Otherwise, this clock can be used, but with a motion defined over the new range. For example, a motion defined from 150 to 270 would animate the clock from 2:30 - 4:30.

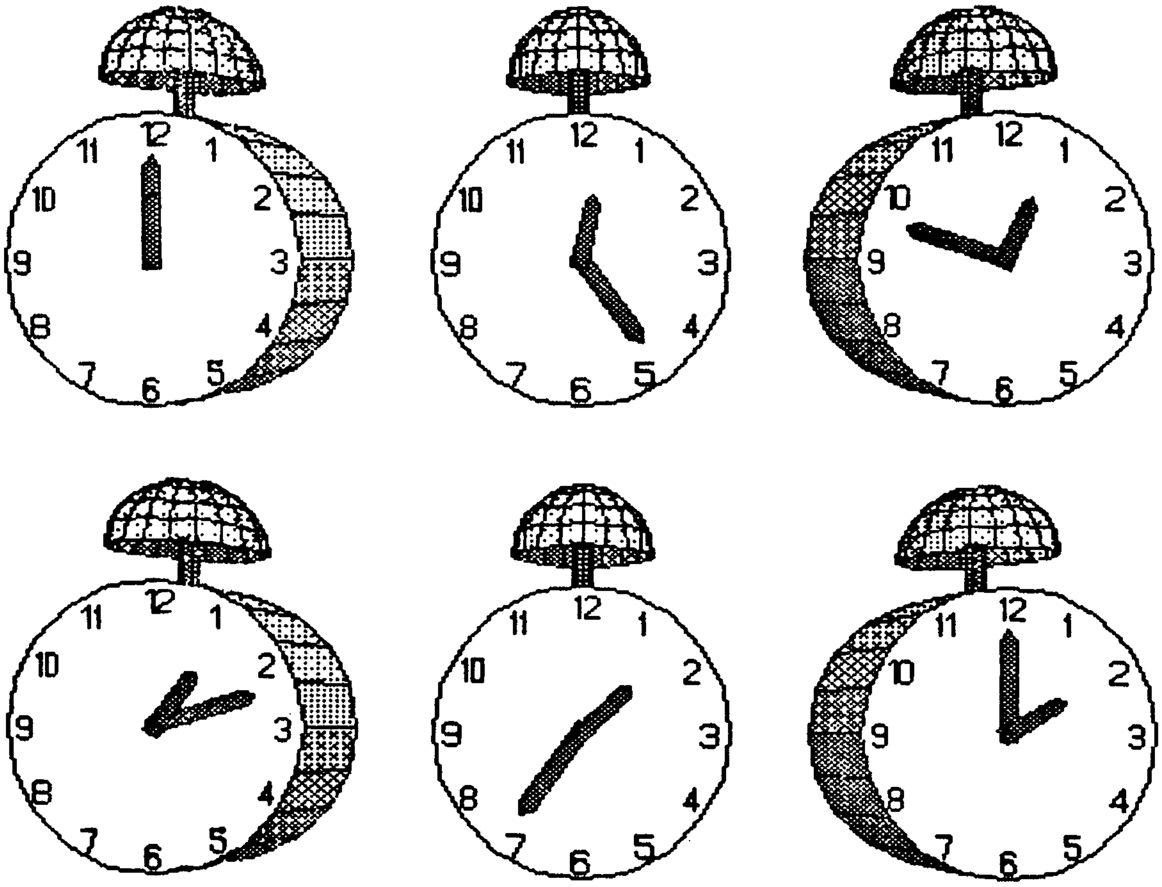

Figure 18

Frames from the clock animation

In the preceding example, the relationship between the joints and motion can be defined mathematically. But the relationship can not always be defined mathematically, consequently, an alternate method for creating motions is provided. The method assumes that the desired motion does not contain a mathematical relationship between the joints and keyframes. It can be tedious to use, nevertheless, it allows the description of complex motions that can not easily be described mathematically. The construct

```
STARTMOTION motion_name
   JOINT joint_id
       [ POSITION angle_x, angle_y, angle_z ]

       FRAME frame_id
          POSITION angle_x, angle_y, angle_z
          INTERPOLATE interpolation_technique

       FRAME frame_id
          POSITION angle_x, angle_y, angle_z
          INTERPOLATE interpolation_technique

       . . .

   ENDJOINT

   JOINT joint_id .....
       FRAME ...

       . . .

   ENDJOINT

   . . .

ENDMOTION
```

allows an exact motion definition. 'motion_name' is a variable of the type MOTION which contains the joint-motion specifications. The joint identifiers correspond to those defined in the animated model. The construct permits a structured approach to produce a motion specification. It does not allow the use of other control constructs. The values of the identifiers must all be constants; no variables are allowed. The keyframe identifier is not used because the construct assumes that the keyframes are placed in an ascending sequential order. The starting position for each joint is specified in the POSITION statement. If the statement is not present in a joint's motion specification, the starting position is assumed to be the neutral position.

4.3 IMPLICIT DEFINITION

Once a motion has been defined, the specification can be viewed as a unit (motion primitive) and it can function as a building block for the definition of more complex motions. A primitive motion algebra has been introduced for this use. The algebra allows the creation of new motions that have been temporally compounded from predefined motions. For example, to create a motion that enables a human model to hop, skip, and jump ten times, the following statement can be used:

```
new_walk = 10 * (hop + skip + jump)
```

In the statement, the symbols 'new_walk', 'hop', 'skip', and 'jump' represent variables of the type MOTION. The symbol '=' is the motion assignment operator. It stores a motion expression on the right hand side to the motion variable on the left hand side. 'new_walk' is the resulting action. 'hop', 'skip', and 'jump' are previously defined motions which allow a model to hop, skip and jump respectively. The constant 10 is the repetition factor that is applied to the actions 'hop', 'skip', and 'jump'.

The symbols '+' and '*' are the operators present in the motion algebra. The '+' addition operator temporally merges two motions into a new motion. The '*' repetition operator creates a new motion by repeating an expression on the right hand side of the '*' by the factor on the left hand side. The '+' and '*' operators can be used interchangeably. Both of the statements

```
new_walk = (hop + skip) + (hop + skip) + (hop + skip)
new_walk = 3 * (hop + skip)
```

produce the same motion. The addition and repetition are independent operations, however, the repetition operator takes precedence over the addition operator. In the statement

```
motion = 10 * hop + skip
```

the motion 'hop' is repeated 10 times before the skipping motion is added.

The motion algebra allows motion expressions of a general form on the right hand side of the '=' assignment operator. The production for the expression is called <motion express>. The Backus-Naur form definition for the production is

```
<motion express> ::= <motion term>
                   | <motion term> + <motion express>

<motion term> ::= <motion factor>
                | <motion factor> <modification list>

<motion factor> ::= <simple>
                  | <simple list>

<simple list> ::= (<motion express>)

<simple> ::= <motion primitive>
           | <repetition>

<repetition> ::= <integer constant> * <repetition list>
               | <integer variable> * <repetition list>

<repetition list> ::= <motion primitive>
                    | <simple list>
```

The motion algebra relies heavily on the availability of predefined motion primitives. It is recognized that the explicit definition of such primitives can be both difficult and time consuming, therefore, operations have been introduced that allow a more convenient definition of the motion primitives. For example, if a motion primitive (walk) exists which allows a model to walk, it may be desirable to employ the action of the legs in a different motion. The operation STRIP has been introduced, which creates a partial motion from a given motion. In the statement

```
walking_legs = STRIP walk, 5, 6, 7, 8, ...
```

'walking_legs' is a new motion primitive that animates a model's legs. The values 5, 6, 7, 8, ..., are the joint identifiers present in the model's legs. Using the method, several motion primitives can be created that animate only portions of a given model. If the preceding statement is executed, the motions in Figure 19 would be present.

MOTION: Walk

Joint 1	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 2	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 3	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 4	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 5	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 6	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 7	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 8	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....

MOTION: Walking_legs

Joint 5	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 6	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 7	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 8	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....

Figure 19

Strip operation

The operation SYNCHRONIZE has also been introduced. When given a set of partial motions, SYNCHRONIZE creates a new motion that animates portions of a model concurrently. The statement

```
my_walk := SYNCHRONIZE walking_legs,
                       swinging_arms
```

defines a motion primitive (my_walk) which allows a figure to walk while swinging its arms. If the preceding statement is executed, the motions in Figure 20 would be present. The use of the STRIP and SYNCHRONIZE operations allow an increase in the number of motion primitives without explicitly defining the motions.

MOTION: Walking_legs

Joint 5	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 6	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 7	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 8	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....

MOTION: Swinging_arms

Joint 1	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 2	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....

MOTION: My_walk

Joint 1	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 2	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 5	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 6	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 7	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....
Joint 8	Starting Position	Keyframe 1	Keyframe 2	Keyframe 3	....

Figure 20

Synchronize operation

4.4 MOTION TRANSFORMATIONS

In addition to the use of the motion algebra, a method has been provided that allows a user to tailor motion primitives to the needs of an animation sequence without knowing details of the underlying motion. Motion elements on the right hand side of the motion assignment represent instances of previously defined motion identifiers. These instances can be modified to create different instances of the same motion with different results. Currently, the modifications permitted are the transformations temporal scaling, and keyframe extreme scaling. They follow a motion in a list separated by angle brackets ('<', '>').

The syntactic form of the transformations is given by

```
SCALE FRAMES <real>
SCALE EXTREMES <real>
```

The transformations are independent, consequently, they are commutative. They may be applied in any order and may be repeated. If a transformation is repeated, the resulting transformation is the product of all the instances.

The SCALE FRAMES transformation temporally scales a motion. Thus, the statement

```
b = a <SCALE FRAMES 2.0>
```

assigns to motion 'b' an instance of motion 'a' that has been temporally stretched to twice its original length. The number of keyframes present in 'b' is the same as in 'a', however, the number of intermediate frames between each keyframe has been doubled. In addition, if the scaling factor is negative, the system will reverse the keyframe order, producing a temporally reversed instance of the motion (Figure 21 and 22).

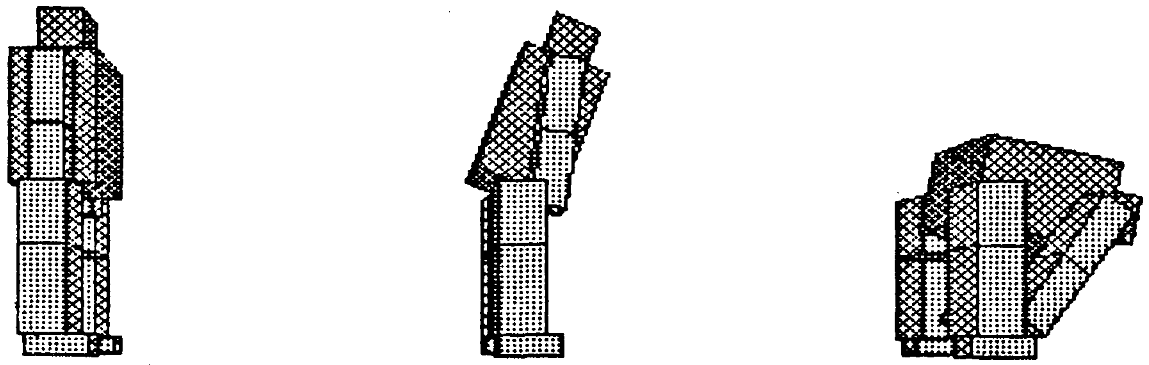

Figure 21

Sample motion a

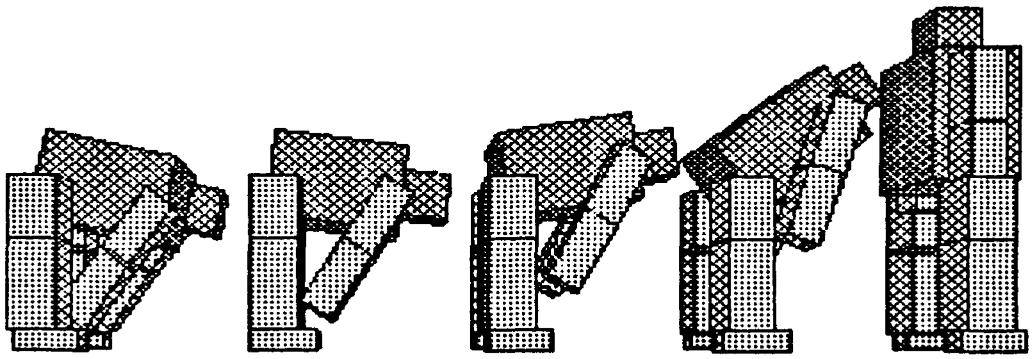

Figure 22

Motion b from b = a <SCALE FRAMES -2.0>

The SCALE EXTREMES transformation scales the keyframe extremes present in a motion. Thus, the statement

```
b = a <SCALE EXTREMES 0.5>
```

assigns to motion 'b' an instance of motion 'a' which has had every keyframe extreme scaled by a factor of 0.5. The number of keyframes present in 'b' remains the same as in 'a', and the number of intermediate frames between each keyframe is unchanged (Figure 21 and 23).

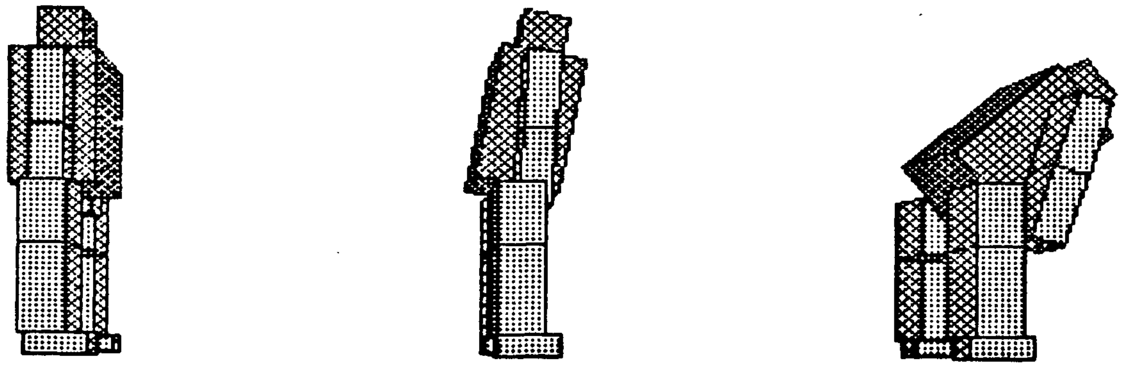

Figure 23

Motion b from b = a <SCALE EXTREMES 0.5>

Such an approach to motion modification gives rise to many transformations that can tailor motion primitives to the needs of an animation sequence. These transformations, as well as the operations '*' and '+', have the advantage that an intimate knowledge of a model's structure and motion definitions are not necessary. They allow motion manipulation at a high level of abstraction.

Chapter 5

CREATING ANIMATION

5.1 TOOLS FOR ANIMATION

Once a model is defined and motions for the model have been created, the model can be animated. A typical animation statement is

```
ANIMATE model_name FROM start_frame TO end_frame
                   USING motion_name [ REPEAT ]
```

where 'model_name' is a predefined articulate model; 'motion_name' is a previously specified motion that is applied to the model. The frame identifiers 'start_frame' and 'end_frame' are temporal endpoints of the model's animation. They are defined relative to the beginning of a scene.

The REPEAT keyword is optional. If the REPEAT is unspecified, the motion is assumed to span the entire animation sequence; the model remains frozen at its last specified orientation, if the motion used does not define movement for the entire time span. The REPEAT is used whenever the motion does not span the entire model animation sequence; the motion is automatically repeated until the time span is covered. Care must be taken in creating motions that are repeated. Severe discontinuities can arise as the motion is finished and restarted. For best results, a model's positional orientation should be similar at both endpoints of the motion.

Once the ANIMATE statement is executed, a graphical object of the name 'model_name' is available to the host language (i.e. the host language treats each model instance present in a frame as a graphical object). The ANIMATE statement produces animation with respect to a model's main link. It is the responsibility of the host language to provide the capabilities to translate the resulting graphical object throughout a scene. LIG ANIMATE provides a set of tools which control the complex motions associated with articulate models, but it does not replace the function of the host language.

Models of the same class (i.e. models of identical structure) and models with different-sized segments react identically when animated by the same motion. Models containing an equal number but different types of joints can still be driven by the same motion specifications, however, the resulting model movements may be difficult to predict. Models place restrictions on their movements. For example, if a rotation is employed that violates a joint's extremes, the joint will enforce its extreme rotation limit over the motion specification given. This restriction permits two models of the same class, but different restrictions, to behave realistically using the same motion.

Once the elements for a scene have been defined, the scene can be explicitly specified with static and dynamic components. A typical statement creating a scene is

```
INITIALIZE SCENE scene_identifier

    initialize scene environment

STARTSCENE LENGTH scene_length
         [ SAMEFRAME imagesperframe ]

   ANIMATE model_name FROM ....
   ANIMATE any dynamic models

         . . .

   movements which vary with time
   (camera movements, panning,
    zooming, etc.)

         . . .

ENDSCENE
```

where 'scene_identifier' is a unique identifier for the scene instance; 'scene_length' is the number of frames displayed during the scene. The INITIALIZE section of the scene permits a user to specify static information that initializes the environment prior to the execution of a scene. By default, the statements present between the STARTSCENE and ENDSCENE are executed once for each frame generated during the scene. The default can be changed by the SAMEFRAME option; it allows a user to display several images of each model on the frame. This option is useful when real-time output capabilities are unavailable. Each model produces several images per frame, thus allowing easier examination of a model's movements when using hardcopy devices.

The scenes defined are displayed by the statement:

```
SHOOTSCENES
```

The scenes are displayed in sequential order beginning from the lowest scene identifier to the highest one. The use of scene constructs is advantageous because the animation sequence can be broken into a set of independent scenes. Animated films can be created on a piecewise basis, thus simplifying the animation process. The creation of independent scenes also permits changes in the scene ordering without any knowledge of the frame identifiers involved.

A statement has been introduced for use primarily within the SCENE construct. The statement

```
BACKGROUND object1, object2, object3, ...
```

defines a list of graphical objects that are displayed as the background of a scene. Placing the BACKGROUND definition in the scene initialization section defines a static background. If the BACKGROUND statement is placed between the STARTSCENE and ENDSCENE, the user can define a new or modified background for each scene's frame by using graphical objects that are modified throughout the life of a scene.

LIG ANIMATE restricts user access to the internal structure of variables of the type MODEL, however, it is possible to observe a model's internal relationships. The statement

```
PRINT variable1, variable2, ...
```

displays the internal status of variables. Variables of either MODEL or MOTION type may be displayed. Specifying a model's name in a PRINT statement produces a listing of the model's internal node relationships. Preceding the model's name with the symbol '$' produces a status report of the model's symbol table. Specifying a motion's name produces a listing of the keyframe information currently stored in the motion.

A model's internal relationships are displayed when LIG ANIMATE processes its source code because the model's database is stored in the preprocessor. The motion and symbol table information is not displayed until the object code is compiled and run because this information is stored at the host language level. The PRINT statement is normally used independently of constructs, however, the statement may be used within the SCENE construct to obtain the status of a model's symbol table at each scene frame.

5.2 <u>MODEL TRAVERSAL</u>

A model's internal tree structure is completely traversed each time a model is displayed (i.e. once each frame). A model's symbol table is updated with each joint's current positional angle, current velocity, and acceleration prior to the traversal of the model. The symbol table information is obtained from both the motion specification and the interpolation routines.

The traversal algorithm is a recursive post-order routine. It assembles each model's instance (from the extremities inward) with respect to the main link in the model. The resulting model instance is a graphical object that is subsequently displayed by the host graphical language. Recursive routines are employed in the tree traversal because they allow storage of the model's primary-secondary link relationship in the recursion stack. The use of recursion also simplifies the model traversal on models whose main links (roots) have been changed.

5.3 LEVELS OF LANGUAGE USAGE

The LIG ANIMATE language may be used at three different levels of abstraction. The motion algebra, STRIP and SYNCHRONIZE functions, and explicit motion definition reside on different abstract levels with each level more flexible that the level directly below it. The first level is a subset of the second which is, in turn, a subset of the third level. The differentiation by levels allows programmers with different levels of sophistication to use the language. Each level is characterized by the amount of information known by the programmer.

The first level entails the use of the motion algebra. At this level, the user need not know the structure of the motion primitives or the model being animated. The user assumes a model is present which meets the desired need and that a set of motions exist which animate the model. The user may then expand on this motion set to create motions specific to the animation sequence desired. Usage at this level assumes the user has had a programmer create the model and basic motions necessary or that a library of models and motions is accessible.

The second level incorporates the use of the JOINT, STRIP and SYNCHRONIZE operations. At this level, the user must have knowledge of both the motion primitives being used and the model being animated. The user need not know the details of the movements present in each motion's joint specification, however, knowledge of their overall effect is needed. Also, the user must know the joint identifiers used in both the model and the motions. The programmer is removed one step from the details necessary in explicit motion definition. At this level, the user can build on existing submodels to create more complex models and create new motion primitives from portions of those already present.

The third and highest level entails the explicit definition of models and motion; the user knows detailed information of the model's structure and each keyframe of the joint-motion specifications. The third level is the most flexible and allows the definition of the most detailed models and motions. The user knows the extremes of each motion and decides on the nature of the interpolation used to create the intermediate frames. This level is used either directly or indirectly by all users.

Chapter 6

CONCLUSION

Three-dimensional computer animation, in some respects, is similar to clay, puppet, or model animation. In three-dimensional animation, the model, props, and backgrounds are built and painted. A motion picture camera is prepared and one frame of the film is exposed. The animator moves the model a little and another frame is exposed. Repeating this procedure is the process of creating animation. As a result of the persistence of vision, when these single frames are projected at an appropriate rate they appear to blend into movement [21].

The object of this thesis is to give a programmer a set of tools for modelling complex articulate figures and for controlling their movements. In addition, movement synthesizing tools are supplied that decrease much of the work associated with generating explicit motion descriptions for articulate figures. LIG ANIMATE allows the creation of computer models of figures, props and backgrounds. Many copies or instances of a model can be used and they can be manipulated by changing their size, position and orientation in space. LIG ANIMATE removes the burden of representing a figure's segment relationships from the host language. Yet, the host language is allowed access to the graphical object produced by the model at each animation frame. The LIG ANIMATE system does not replace the function of the host language. The system is responsible for a model's movement with respect to itself; the host language remains responsible for the model's movement as a unit. LIG ANIMATE simplifies the work of the host language but still allows the flexibilities associated with the host language.

Many computer animation systems allow a user to describe motions for three-dimensional objects by geometric descriptions. Defining a realistic walking sequence is difficult because the graphics must be controlled by the motion dynamics of the walking object [1]. Other systems approach the problem by assigning constraints to the limbs that relate to the constraints found in nature and by using goal-directed techniques to create separate movements which are part of the final motion [23]. These methods tend to be computationally expensive and usually isolate the user from both the models and motions.

LIG ANIMATE has a simpler approach to the problem of animation specification which falls between the extremes of motion dynamics and goal direction. It attempts to raise the level of animation specification by extending the structure present in high-level graphical languages. One strength associated with LIG ANIMATE's approach is flexibility. For example, the interpolations are defined independently of the LIG ANIMATE preprocessor. A programmer can use existing interpolation subroutines, or can add new ones. Interpolation subroutines that are more sophisticated can easily be added to LIG ANIMATE, thus allowing more realistical model movement. Interpolations can also be created that check for specific events, thus allowing animation synchronization by message passing between models, without modifying the LIG ANIMATE preprocessor.

Several animation systems have attempted to approach animation from a high level of abstraction. Directors and scripts are employed [5, 14], however, the articulate models are implemented as procedures (i.e. the model and motion are inseparable). Such systems lack the ability to add animation sequences, or to change the motions associated with a model. They also do not permit the transformation of model instances. For example, if a procedure has been created which animates a man walking, a separate procedure must be written to animate a child walking, even though the structure of both models is identical.

LIG ANIMATE allows a structured approach to the creation of both models and motions. Models and motions can be coarsely defined and subsequently refined to a desired level. For example, a walking motion can initially be specified that primarily relies on the use of a model's hip and shoulder joints; the rest of the model remains rigid. Once the hip and shoulder movements are correct, the movements associated with the secondary joints (elbows, knees, etc.) can be specified. This approach simplifies the creation of motion because it is possible to deal with a motion's movements individually.

The same approach can be applied to model development. A model may be defined with only the major joints and as the development proceeds more joints can be added. Another approach to model development is also possible. A model's internal structure can be specified independently of the graphical objects that represent the segments, therefore, the graphical objects can be modified to redefine a model's appearance. For example, a block may initially represent a segment; the segment can subsequently be represented by more complex polygonal structures. This segment development can proceed independently of a model's definition, provided the dimensions of the graphical objects remain essentially the same.

LIG ANIMATE's approach to defining motion is flexible because data generation software can create motion specifications. Human motion specification can be simplified by deriving real time data from instrumentation. A human can be wired with devices (electrogoniometers) that measure positional orientation [8]. The movements performed by the human can be recorded and subsequently used to animate human models.

As is the case with most projects, further work would be beneficial. LIG ANIMATE currently restricts joints to rotational joints (i.e. the secondary segment rotates about a pivotal point located on the primary segment). Prismatic joints (i.e. a joint in which the secondary link slides within the primary link) are commonly used in robotic arms. LIG ANIMATE could be extended to permit the use of all joints, rather than permitting only rotational joints. Further work could permit the system to use segments which change in size and shape throughout a model's animation sequence. The segment's shape and size could be defined using a keyframe technique; the language would in-between the appropriate segments during the animation. Such an ability would allow the metamorphosis of models (e.g. a mail box could change into a human during a scene).

LIG ANIMATE is designed as an extension to a host language, and to be executed in a batch environment. A preprocessor translates all extended language statements into procedure calls. Other implementations, such as a command language that is interpreted by an executive kernel, are possible. The batch environment complicates the use of predefined models and motions. At present, these definitions must be made at the beginning of an application program before they can be employed. Most computer animation studios build up the equivalent of a shopping catalog of models that are available for future work [10]. Ideally, there should be a method to archive the models and motions defined in a program. The models and motions would be assigned names and stored in a library. Subsequent programs would need only load definitions for the models and motions before using them. Such an approach would increase the flexibility and speed associated with the creation of animation using LIG ANIMATE.

The use of library definitions creates a new problem. Unless a user has specified a model and the associated motions, or has previously worked with them, it is difficult to determine a model's potential movements. It would be useful to have a viewing utility that permits the viewing of predefined models and motions in an interactive environment. This technique would allow the user to determine exactly what had been previously defined and what further definitions must be made in the application program.

LIG ANIMATE has several advantages over existing systems. It permits the modelling and animation of figures at a high level of abstraction from within a programming language thus allowing the implementation of sophisticated application programs that are easy to read and understand. Where many systems can not easily work with rotations, LIG

ANIMATE deals with rotations effectively. The use of rotations enables the creation of generalized motions which can be applied to models of the same joint structure; motion specifications using paths and forces can only be used on the specific models for which they were designed. Articulate figures are frequently interpolated at the point level because a two-dimensional projection of the figure is used. The approach presented permits the modelling of three-dimensional figures and their subsequent interpolation on a rotational basis (i.e. the rotational extremes correspond to the keyframe drawings in two-dimensional interpolation), thus allowing for easier specification of three-dimensional animation.

BIBLIOGRAPHY

[1] Armstrong, W.W., and M. Green, The Dynamics of Articulated Rigid Bodies for Purposes of Animation, Graphics Interface '85, 1985, pp. 407-415.

[2] Badler, N.I., J. O'Rourke, and B. Kaufman, Special Problems in Human Movement Simulation, Computer Graphics, Vol. 14, No. 3, 1980, pp. 189-197.

[3] Badler, N.I., and S.W. Smoliar, Digital Representations of Human Movement, ACM Computing Surveys, Vol. 11, No. 1, 1979, pp. 19-38.

[4] Baecker, R.M., Picture-Driven Animation, AFIPS Conference Proceedings, Vol. 34, 1969, pp. 273-288.

[5] Bergeron, P., A Structured Motion Specification in 3D Computer Animation, Graphics Interface '83, 1983, pp. 215-222.

[6] Booth, K.S., D.H. Kochanek, and M. Wein, Computers Animate Films and Video, IEEE Spectrum, Vol. 20, No. 2, 1983, pp. 44-51.

[7] Burtnyk, N., and M. Wein, Computer-Generated Key-Frame Animation, Society of Motion Picture + TV Engineers, Vol. 80, 1971, pp. 149-153.

[8] Calvert, E., J. Chapman, and A. Patla, The Simulation of Human Movement, Graphics Interface '82, 1982, pp. 227-234.

[9] Catmull, E., The Problems of Computer-Assisted Animation, Computer Graphics, Vol. 12, No. 3, 1978, pp. 348-353.

[10] Chuang, R., G. Entis, 3D Shaded Computer Animation, Step-by-Step, Computer Graphics - Theory and Applications, Springer-Verlag, Tokyo, 1983.

[11] Horn, B.K.P., Kinematics, Statics, and Dynamics of Two-Dimensional Manipulators, Artificial Intelligence: An MIT Perspective, Vol. 2, 1979.

[12] Korein, J.U., and N.I. Badler, Techniques for Generating the Goal Directed Motion of Articulated Structures, IEEE Computer Graphics and Applications, Vol. 2, No. 9, 1982, pp. 71-81.

[13] Magnenat-Thalmann, N., and D. Thalmann, Actor and Camera Data Types in Computer Animation, Graphics Interface '83, 1983, pp. 203-207.

[14] Magnenat-Thalmann, N., and D. Thalmann, Director-Oriented 3D Shaded Computer Animation, Graphics Interface '84, 1984, pp. 1-7.

[15] McGhee, R.M., Control of Legged Locomotion Systems, Proc. 1977 Joint Automatic Control Conference, Vol. 1, pp. 205-213.

[16] McKeeman, W.M., J.J. Horning, and D.B. Wortman, A Compiler Generator, Prentice-Hall, Englewood Cliffs, 1970.

[17] Paul, R.P., Robot Manipulators, MIT Press, Cambridge, Mass., 1981.

[18] Perez, L., and M.A. Wesley, An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles, Communications of the ACM, Vol. 22, 1979, pp. 560-570.

[19] Ross, R., A High-level Graphics Programming Language Supporting the Inquiry of Graphical Objects, M.A.Sc. Thesis, The University of British Columbia, 1982, 88 pp.

[20] Ross, R., LIG6: Language for Interactive Graphics, User's Manual, Department of Electrical Engineering, The University of British Columbia, 1982, 55 pp.

[21] Sachter, J., 3-D Computer Generated Animation, Proc. 4th Symposium on Small Computers in the Arts, 1984, pp. 42-49.

[22] Schrack, G.F., Design, Implementation and Experiences with High-level Graphics Language for Interactive Computer-Aided Design Purposes, Computer Graphics, Vol. 10, No. 1, 1976, pp. 10-17.

[23] Zeltzer, D., Motor Control Techniques for Figure Animation, IEEE Computer Graphics and Applications, Vol. 2, No. 9, 1982, pp. 53-59.

[24] Zeltzer, D., Representation of Complex Animation Figures, Graphics Interface '82, 1982, pp. 205-211.

APPENDIX A

A LIG ANIMATE PROGRAM

The LIG ANIMATE program shown on the following pages was used to create the animation sequence present in Figure 11. PRIMITIVE BOX is a LIG6 graphical subprogram which creates graphical objects for the figure's segments. The statements flagged with an asterisk in the first column are processed by the LIG ANIMATE preprocessor. The host statements are copied to the preprocessor target file. This program demonstrates the ability of LIG ANIMATE to animate an articulate figure while allowing the host language access to the figure as a unit.

```
C       PRIMITIVE BOX produces a 3D rectangle of arbitrary
C       height, width and depth.  The rectangle is defined
C       about the origin.  Width is along the X axis.  Height
C       is along the Y axis.  Depth is along the Z axis.

        PRIMITIVE BOX
        REAL HEIGHT, WIDTH, DEPTH, LIGHT, LIGHT2
        VECTOR CENTRE, FTR, FTL, FBR, FBL, BTR, BTL, BBR, BBL

        BOX ::= 'BOX' ['AT' CENTRE ',']
                      <CENTRE = (0.0, 0.0, 0.0)>
                      'WIDTH'  WIDTH  ','
                      'HEIGHT' HEIGHT ','
                      'DEPTH'  DEPTH
                      [',' 'LIGHTNESS' LIGHT]<LIGHT = 40.0> ;

        LIGHT2 = LIGHT * 1.5

        FTR = CENTRE + (WIDTH/2, HEIGHT/2, DEPTH/2)
        FTL = FTR - (WIDTH, 0.0, 0.0)
        FBL = FTL - (0.0, HEIGHT, 0.0)
        FBR = FTR - (0.0, HEIGHT, 0.0)
        BTR = FTR - (0.0, 0.0, DEPTH)
        BTL = BTR - (WIDTH, 0.0, 0.0)
        BBL = BTL - (0.0, HEIGHT, 0.0)
        BBR = BTR - (0.0, HEIGHT, 0.0)

C       Set the drawing parameters

        DRAW WITH <LIGHTNESS LIGHT>

C       Back

        DRAW POLY FROM (BTR) TO (BTL) TO (BBL) TO (BBR)
                             TO (BTR)

C       Set the drawing parameters

        DRAW WITH <LIGHTNESS LIGHT2>

C       Top

        DRAW POLY FROM (FTR) TO (BTR) TO (BTL) TO (FTL)
                             TO (FTR)

C       Bottom

        DRAW POLY FROM (FBR) TO (BBR) TO (BBL) TO (FBL)
                             TO (FBR)

C       Left

        DRAW POLY FROM (FTL) TO (BTL) TO (BBL) TO (FBL)
                             TO (FTL)
```

```
C     Right

      DRAW POLY FROM (FTR) TO (BTR) TO (BBR) TO (FBR)
                           TO (FTR)

C     Set the drawing parameters

      DRAW WITH <LIGHTNESS LIGHT>

C     Front

      DRAW POLY FROM (FTR) TO (FTL) TO (FBL) TO (FBR)
                           TO (FTR)

      RETURN
      END


      GRAPHICAL HEAD, BODY, TORSO
      GRAPHICAL FARM, BARM, FLEG, BLEG, FOOT
      VECTOR    LOCATN, DISTAN
      REAL      VUWID, TORWID, FRMNUM

*     MODEL HUMAN, LARM, RARM, LLEG, RLEG
*     MOTION SQUAT

C     Define the model's HEAD

      HEAD :- BOX WIDTH 0.30, HEIGHT 0.30, DEPTH 0.16

C     Define the model's TORSO

      BODY :- BOX WIDTH 0.56, HEIGHT 0.86, DEPTH 0.36
      TORSO :- (HEAD + BODY<TRANS(0.0, -0.58, 0.0)>)
               <TRANS(0.0, 0.58, 0.0)>

C     Define the model's limbs
C     (Forearm, Backarm, Foreleg, Backleg)

      FARM :- BOX WIDTH 0.22, HEIGHT 0.48, DEPTH 0.14,
                  LIGHTNESS 25.0
      BARM :- BOX WIDTH 0.22, HEIGHT 0.48, DEPTH 0.14,
                  LIGHTNESS 25.0
      FLEG :- BOX WIDTH 0.30, HEIGHT 0.58, DEPTH 0.18,
                  LIGHTNESS 25.0
      BLEG :- BOX WIDTH 0.30, HEIGHT 0.40, DEPTH 0.18,
                  LIGHTNESS 25.0

C     Define the model's FOOT

      FOOT :- BOX WIDTH 0.42, HEIGHT 0.13, DEPTH 0.10,
                  LIGHTNESS 25.0
```

```
C     Define the model's right and left arms

*     RARM :- JOINT 1, BARM, (0.0, -0.24, 0.0),
*                      FARM, (0.0, 0.24, 0.0),
*                      (0.0, 0.0), (0.0, 0.0), (0.0, 135.0)

*     LARM :- RARM <TRANS JOINTS BY 1>

C     Define the model's right and left legs

*     STARTMODEL RLEG
*         JOINT 3, BLEG, ( 0.00, -0.20, 0.0),
*                  FLEG, ( 0.00, 0.29, 0.0),
*                  (0.0, 0.0), (0.0, 0.0), (-150.0, 0.0)
*         JOINT 4, FLEG, (-0.05, -0.29, 0.0),
*                  FOOT, (-0.11, 0.06, 0.0),
*                  (0.0, 45.0), (-25.0, 25.0),(-50.0, 50.0)
*     ENDMODEL

*     LLEG :- RLEG <TRANS JOINTS BY 2>

C     Add the extremities to the model HUMAN

*     HUMAN :- JOINT  7, TORSO, (0.00, 0.40,-0.18),
*                        RARM, (0.0, 0.24, 0.0),
*                        (0.0, 180.0), (-20.0, 20.0),
*                        (-90.0, 180.0)
*     HUMAN :- JOINT  8, HUMAN, (0.00, 0.40, 0.18),
*                        LARM, (0.0, 0.24, 0.0),
*                        (0.0, 180.0), (-20.0, 20.0),
*                        (-90.0, 180.0)
*     HUMAN :- JOINT  9, HUMAN, (0.00,-0.40,-0.18),
*                        RLEG, (0.0, 0.24, 0.0),
*                        (-45.0, 45.0), (-45.0, 45.0),
*                        (-90.0, 135.0)
*     HUMAN :- JOINT 10, HUMAN, (0.00,-0.40, 0.18),
*                        LLEG, (0.0, 0.24, 0.0),
*                        (-45.0, 45.0), (-45.0, 45.0),
*                        (-90.0, 135.0)

C     Define the motion SQUAT

*     STARTMOTION SQUAT
*         JOINT  1 POSITION 0.0, 0.0, 135.0
*         JOINT  2 POSITION 0.0, 0.0, 135.0
*         JOINT  3 POSITION 0.0, 0.0, 0.0
*             FRAME 5 POSITION 0.0, 0.0,-135.0
*                     INTERPOLATE ACCELERATE
*         JOINT  4 POSITION 0.0, 0.0, 0.0
*             FRAME 5 POSITION 0.0, 0.0,  45.0
*                     INTERPOLATE ACCELERATE
*         JOINT  5 POSITION 0.0, 0.0, 0.0
*             FRAME 5 POSITION 0.0, 0.0,-135.0
*                     INTERPOLATE ACCELERATE
*           JOINT  6 POSITION 0.0, 0.0, 0.0
*               FRAME 5 POSITION 0.0, 0.0,  45.0
*                       INTERPOLATE ACCELERATE
*           JOINT  9 POSITION 0.0, 0.0, 0.0
*               FRAME 5 POSITION 0.0, 0.0,  90.0
*                       INTERPOLATE ACCELERATE
*           JOINT 10 POSITION 0.0, 0.0, 0.0
*               FRAME 5 POSITION 0.0, 0.0,  90.0
*                       INTERPOLATE ACCELERATE
*       ENDMOTION

*       FIGURE HUMAN MAIN LINK FOOT

*       INITIALIZE SCENE 1

            VIEW POINT = (0.0, 0.0, 4.0)
            VIEW WIDTH = 7.5

            DISTAN  = ( 6.0, 0.0, 0.0)
            LOCATN  = (-2.0, 0.0, 0.0)

*       STARTSCENE LENGTH 6 SAMEFRAME 6

*           ANIMATE HUMAN FROM 0 TO 5 USING SQUAT

            HUMAN :- HUMAN <TRANS (LOCATN)>

            LOCATN = LOCATN + (DISTAN / 5.0)

*       ENDSCENE

*       SHOOTSCENE

        STOP
        END
```

APPENDIX B

IMPLEMENTATION NOTES

The language LIG ANIMATE is implemented on the University of British Columbia Computing Centre's Amdahl 5850 computer under the Michigan Terminal System. The following MTS commands will execute a LIG ANIMATE program contained in the file LIGPROGRAM.

```
$RUN LIGANIMATE.O SCARDS=LIGPROGRAM SPRINT=-OUTPUT
                  SPUNCH=-LIGCODE PAR=NEW=35 LIST=-LIST
$RUN LIG6:LIG6 SCARDS=-LIGCODE SPUNCH=-FTN
$RUN *FTN SCARDS=-FTN SPUNCH=-FTNOBJ
$RUN -FTNOBJ+LIG6:LIB+LIGANIMATE.LIB
```

Several statistics regarding the implementation of LIG ANIMATE have been obtained. The preprocessor was written in PASCAL using a top-down Compiler Writing System. The resulting preprocessor consists of 229 procedures totalling 6,312 lines of code. This represents a listing of 124 pages and occupies a disk file containing 56 pages (each disk page contains 4096 bytes). The object code resulting from the compilation of the preprocessor source code requires a disk file of 50 pages and it takes 0.334 seconds of CPU time to load.

The run-time library was written in LIG6; it contains 28 subroutines consisting of 1,151 lines of code. This represents a listing of 22 pages and occupies a disk file of 12 pages. The resulting object code requires a 12 page disk file and takes 0.083 seconds of CPU time to load.

The translation of a LIG ANIMATE program into an equivalent LIG6 program involves the replacement of extension constructs with LIG6 statements and calls to subroutines in a run-time library. The actual expansion that occurs depends upon the program being processed. In the program shown in Appendix A, the 99 statement program took 1.139 CPU seconds and cost $0.29 for the preprocessor to analyse the program, create an equivalent 234 statement LIG6 program, and produce a listing. The expansion factor for the LIG ANIMATE program was 2.4. The HUMAN model was defined using 11 statements. The equivalent LIG6 code contains 75 statements, an expansion factor of 6.8. The SQUAT motion was defined using 16 statements. The equivalent LIG6 code contains 29 statements, an expansion factor of 1.8.