BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-90-M6

Articulated Objects in BAGS

by
Lisa Kay Borden

Articulated Objects in BAGS

by

Lisa Kay Borden

B.A., Mills College, 1984

Research Project

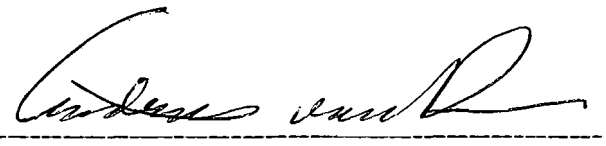
Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University.

May 1990

This project by Lisa Kay Borden

is accepted in its present form by the Department of Computer Science

as satisfying the project requirement for the degree of Master of Science

Date 5/9/90 

Andries van Dam

Acknowledgements

I would like to thank Professors Andries van Dam and John Hughes for providing a unique and stimulating environment in which to study computer graphics at Brown University. I am indebted to Henry Kaufman for a key idea used in this work and many helpful suggestions along the way. Scott Snibbe, Brook Conner and Dan Robbins contributed important interface and design ideas as well as artistic balance. A special thank you goes to Bob Zeleznik for endless debugging support and advice beyond the call of duty. I am also grateful to all the members of the Brown Computer Graphics Group for their good ideas, good music and good company.

This work has been sponsored in part through research contracts with International Business Machines Corporation, National Cash Register and Digital Equipment Corporation, and equipment grants from Sun Microsystems and Hewlett-Packard.

Articulated Objects in BAGS

Lisa Kay Borden

April 2, 1990

Abstract

The current aim of research in the Computer Graphics Group at Brown University is to explore interactive modeling and animation techniques using the Brown Animation Generation System (BAGS). This paper describes the implementation of a new modeling technique for articulated objects, and high level motion control methods used to produce constrained motion of the articulated objects.

Contents

1 Introduction

Over the past several years, the Computer Graphics Group at Brown University has developed an interactive software testbed for research in computer graphics and animation. This system, the Brown Animation Generation System (BAGS) [STRA88], is used by artists and programmers to model scenes consisting of complex objects which change over time, producing animations.

The current focus of the group is to enhance the interactive nature of BAGS so that it is easier for the user, whether classical animator or programmer, to create sophisticated animations. With this goal in mind, it is necessary to provide powerful tools that make both modeling and animation accessible and intuitive.

The tools necessary for interactive animation are diverse. There must be an interactive interface for creating both objects and changes over time. A variety of methods for creating complex objects is essential for the animator to create an interesting scene. For animation, both simple transformations and complex motion control methods will enhance the power of the animator. The interface to all of these tools must be easily understood and usable immediately by a novice for at least basic animations.

In this paper, we will discuss the contribution to interactive animation provided by the research and implementation of articulated objects in BAGS. Articulated, or jointed, objects occur in many situations and are natural targets for modeling. Human figures and mechanical linkages are common examples of objects that an animator might wish to model and animate with constrained, correct motion.

Throughout the discussion, we will refer to the individual moving parts in an articulated figure as *links,* or *link objects;* and to connections between two links in an articulated figure as *joints,* or *joint objects.* A collection of links and joints assembled together into a complex object will be referred to as a *linkage.* The set of routines implementing the articulated object model is known as the *LINK package.*

First, motivation is given for the addition of such objects and their corresponding motion-control methods. Related systems which contributed to the design of the BAGS articulated object model are described. A description of the model and its implementation, with the general algorithms used, is given. A brief comparison of several available inverse kinematics algorithms follows. Finally, conclusions drawn from the research and future directions

for this work are presented.

2 Motivation

Currently, BAGS provides a software environment for producing a high-quality image, or series of images that create an animation. A scene is created by modeling a set of 3 dimensional objects, then adding lights and cameras to illuminate and view the objects. Animation occurs when these objects change over time. Thus, creating an image in BAGS consists of the modeling phase, and, optionally, an animation specification phase.

Both the modeling of objects and their animation are described using the SCEne FOrmatting (SCEFO) language [CONN89] implemented as part of BAGS. SCEFO is the basis for all images produced in BAGS. The series of SCEFO statements which create a scene and animate it is known as a script.

Until recently, the process for modeling and animation of scenes using SCEFO was always accomplished in distinctly separate phases. For modeling, there were two separate choices: the user could simply write the script in a text editor, if she were familiar enough with SCEFO to accurately model the types of objects desired in the scene. To view the image, the SCEFO script would be rendered and the user would then make any necessary modifications to the script, re-render the image, and iterate this process until the image was exactly as desired. While certainly functional, this non-interactive approach provided no immediate feedback as to the correctness of the script, and it also put the burden of writing SCEFO onto the animator. Since SCEFO is essentially a programming language, the animator was also required to be a programmer.

Alternatively, The Interactive Modeler (TIM) software system could be used to model the objects without actually having to write the SCEFO statements by hand. TIM provided an interface which allowed users to create objects, position them, change their characteristics, alter any modeling parameter allowed in SCEFO, and then see the results interactively. The user is not required to write any SCEFO since the script is generated by TIM. The immediate visual feedback also cuts down the time necessary for modeling, since changes can be made and viewed at the same time.

When the user turned to the animation phase, however, skill as a programmer was again necessary. The user needed to add SCEFO statements

to the script to produce animation, re-render the script, view the animation, and then modify the script if necessary. This iterative process presents the same problems as the iterative modeling option: it is tedious, non-interactive, and requires that the animator also be a programmer.

The importance of an interactive animator has been clearly shown by the limitations of the above approaches. At this time, an interactive animator, MOO, has been implemented for BAGS and is being developed into the powerful tool necessary for creating the sophisticated types of animations that are supported in SCEFO. While not all features of SCEFO are currently built into MOO, they will be added as time permits. MOO is supported by an underlying interactive scene database (DBOSS), which allows multiple client programs to interact with the same scene.

The interactive animator provides an excellent research testbed whose flexibility and extensibility are readily exploited. There are at least two ways to do this: 1) build into MOO, and its underlying language, SCEFO, additional methods of modeling complex objects; and 2) provide animation methods, known as high-level motion control, to animate these complex objects. The work described in this paper was in large part motivated by these two goals.

2.1 Creating Complex Objects

An articulated object such as a human skeleton represents a complex object built up out of simpler objects. If we consider each bone in the skeleton as a simple object, we construct the entire articulated figure by linking the simple bone objects with joints to create an overall figure. The joints express two different types of constraints: a connectivity constraint, which maintains a specified distance between links connected by the joint; and a degree of freedom constraint, which allows rotation or translation only about a particular axis or set of axes appropriate to the joint.

In constructing a model for an articulated object, it is necessary to embed at least these simple constraints into the model. We will examine both previous work at Brown and other models to see how this information is handled by related systems.

2.2 High-Level Motion Control

In addition to a constrained, yet simple model for jointed objects, the focus on interactive animation at Brown has created an interest in high-level motion control methods. High-level or automatic motion control, referred to as task-level animation in [ZELT85], is the process of abstracting the motion in an animation into higher level descriptions than the simple transformations that are available in SCEFO. For example, an animator might wish to specify that a mechanical robot arm "reach" for a goal object, rather than specifying the series of joint rotations that would cause the arm to reach the goal. The specification of a set of joint rotations for an articulated object is known as *forward kinematics;* the "reach" problem is termed *inverse kinematics,* since the user is only required to specify the goal, rather than the transformations necessary at each joint [JASM89]. With the ability to model articulated objects comes a need for easy specification of both forward and inverse kinematics. Both of these methods can be termed high-level motion control, since internal computation is necessary to translate the user's abstraction of motion into the correct constrained motion for all objects in the linkage. We will term the high-level motion control methods used in BAGS *simulators.*

3 Related Work

Research into and implementation of models for articulated objects in computer graphics and animation has resulted in many sophisticated systems. In this section, we examine those systems which influenced the model implemented by this project. The two issues of importance to look at in related systems are 1) non-redundant, intuitive input specification of the joint/linkage information by the user; and 2) efficacy of the motion-control methods associated with these systems.

The first consideration is extremely important if the joint model is going to be accessible to animators. As described by [RAFE88], and expanded upon by [BEYE88], specifying joints can involve a large amount of input. Each joint has potentially six degrees of freedom (DOFs) – three rotational and three translational. For each joint, a position and orientation must be specified. For each degree of freedom within a joint, allowable constraints

such as joint limits must also be specified.

In addition to minimizing the input necessary for each joint in a linkage, it is desirable that the user not be required to specify any other connectivity information than the joints themselves. For example, if the user creates joint specifications for a shoulder, elbow and wrist in a human arm, she should not also have to specify a grouping of the links in the arm so that the arm can be moved as one unit. This capability should be automatically built into the model. This problem has not previously been solved in BAGS and the automatic grouping and maintaining of these connectivity constraints is a major contribution of this project.

The second issue in existing systems concerns how the jointed objects are animated. A usable system should support primitive transformations for both *direct manipulation* and, as mentioned previously, some sort of *goal-directed or high-level motion.* These methods must perform at real time speeds in order to be useful in an interactive setting. This criterion will be discussed in Section 5.

3.1 Existing Work in BAGS

The addition of jointed objects to BAGS has been discussed and researched by several people [RAFE88], [BEYE88], [MOTT88]. This project is indebted to the work of Rafey [RAFE88] for many conclusions on the best way to model joints.

Rafey points out that the amount of input necessary for a joint specification can grow exceedingly large as the number of joints grow. He solves this problem by classifying joints into 9 distinct types (Figure 1). This classification, taken from [ISAA87], eliminates the need for input specifications for unused degrees of freedom. Many joint relationships can be specified by a simple pin joint, a one degree of freedom rotational joint. There is no need for the user to specify initial angle values or limit constraints for the other 5 degrees of freedom. The classification of joints by number and type of DOFs relieves the user from having to input irrelevant information. Accordingly, the ability to classify joints by type has been included in the current implementation of the joint model.

Beyer [BEYE88] points out that the classification of joints can be extended from nine types of joints to sixteen (Figure 2). He models these joints as *constraints;* each joint has from 0 to 3 constraints on the rotational

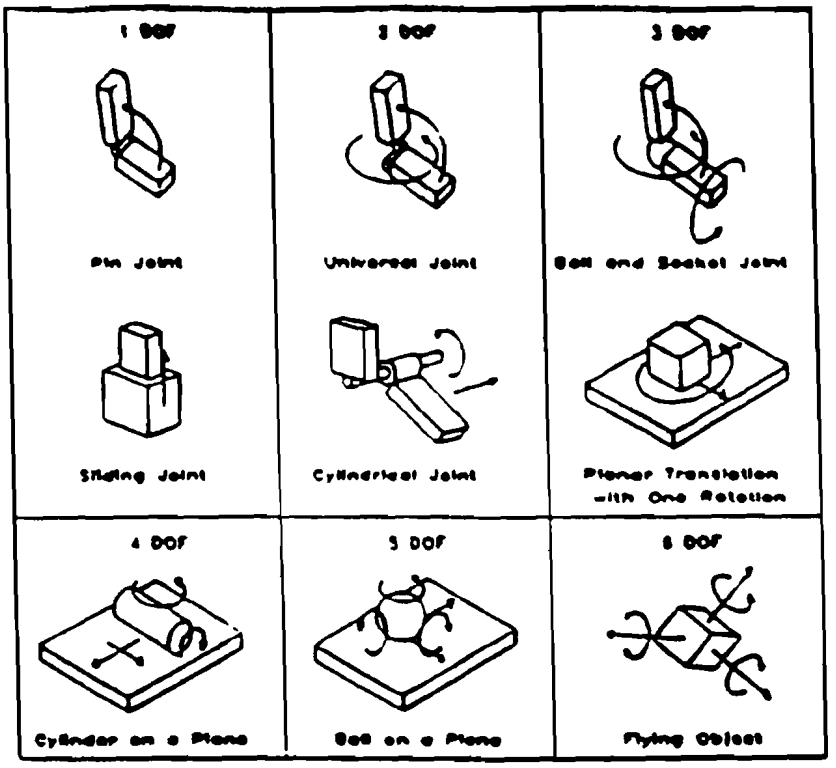


Figure 1: 9 common joint types

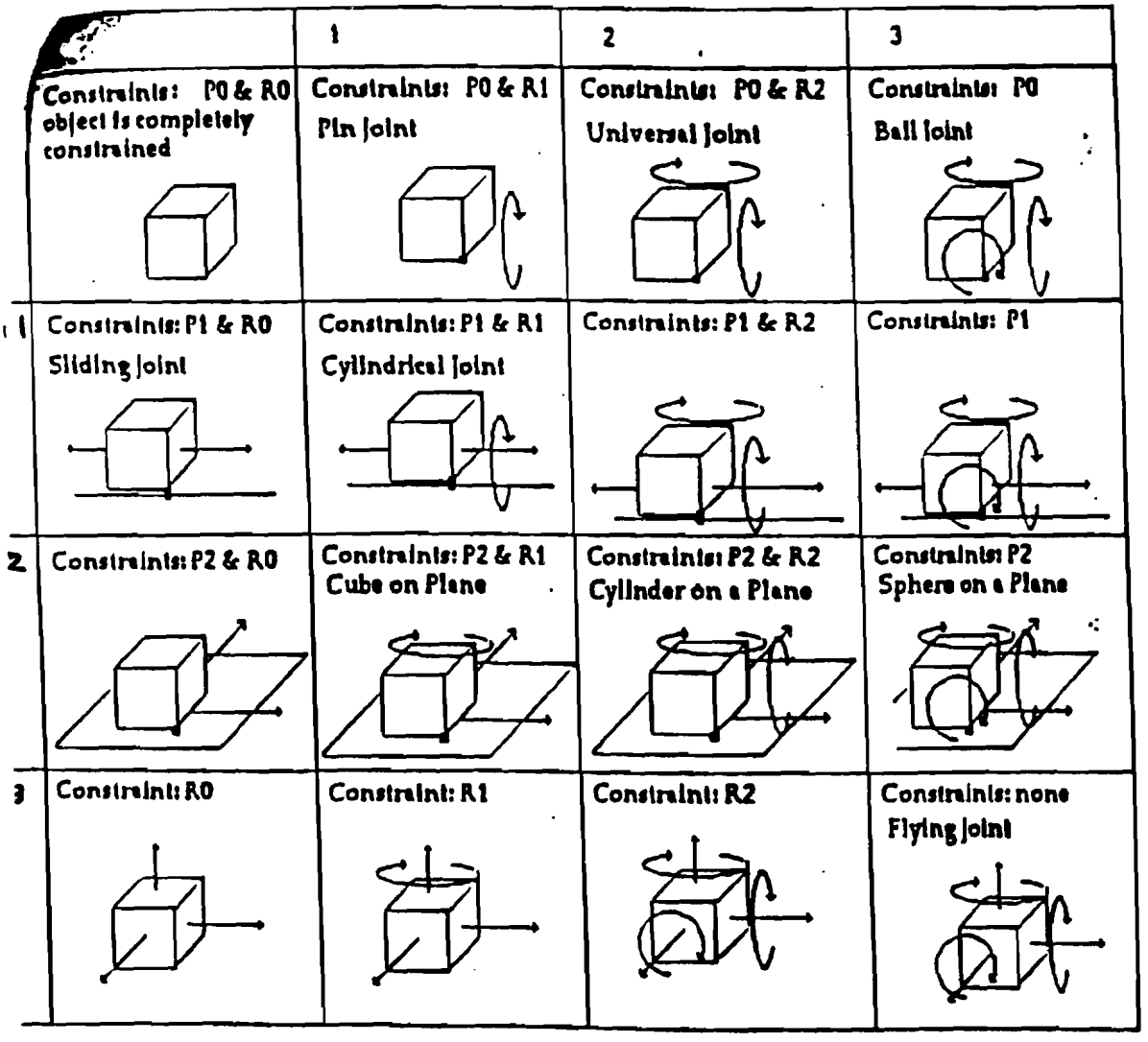


Figure 2: joints modeled as constraints, with extensions to 16 joint types

and translational degrees of freedom. In reality, there are more than sixteen combinations since the axes of motion in Beyer's model are assigned somewhat arbitrarily. For instance, the type of joint labeled "Cube on Plane" is a combination of two degrees of translational freedom and one degree of rotational freedom. The translational axes are arbitrarily assigned to be the X and Z axes in the joint space, while the rotational axis is the Y axis. There is no reason why the translational plane cannot be specified by another combination of 2 axes, such as the X and Y axes, and thus the rotational axis becomes the Z axis.

The sixteen classifications specified by Beyer were incorporated into our model as a convenient first attempt at providing users with the type of joints they most commonly use. As the system is utilized, users may find that they want other combinations of degrees of freedom not available, and those types of joints should be incorporated. Since the internal mechanism for building up a joint from several DOFs is flexible, this should be a simple addition to the system.

Rafey's model for joints solves the redundant input problem using classification of joints by their degrees of freedom. However, it does not maintain any connectivity constraints that are inherently expressed by the existence of the joints. Users can create joints, and can rotate objects about those joints, but the user is responsible for rotating all the right objects about each joint. For instance, if a shoulder joint is created, there is no automatic way for the user to rotate the entire arm about the shoulder. A separate complex object for the arm must be manually created. If she then specifies an elbow joint, she must also create a complex object for the group of objects that exist in the lower arm, and which rotate about the elbow. In order to account for all possible sets of objects that might rotate about n joints in a linkage, $2n$ complex objects would need to be specified. This can be cumbersome for even a small value of n.

To avoid introducing this complexity, it is desirable for the simulator code handling the linkages to maintain the object dependencies present at each joint. However, making these dependencies internal to the simulator does present one problem. For any given joint there are two sets of objects possible for motion. At a knee joint, the transformation might apply to the lower leg and foot, or it might apply to the upper leg and its connected parts. The notion of a current direction for joint transformations is necessary, and this direction can be viewed as a parent/child relationship between the joint

and its two connecting links. At any given time, one link is the parent and one link is the child. When a transformation is specified at a joint, it is the child link and its connected parts which will be transformed. If the user then desires to transform the opposite link and its connected parts, the parent/child relationship must be reversed.

Rafey solves this problem by having the input from the user include a reference to the moving, or child, link, as well as a reference to the joint. This input is sent to a pre-defined SCEFO action for the type of joint desired, and the proper links are then updated as well as the joint axes' position and orientation.

The approach taken by the current implementation is slightly different. It is based on the concept of a *rooted* linkage, such as that used by the Jack system at the University of Pennsylvania [PHIL89]. The idea of rooting a linkage, and its relation to the internal representation of the linkage as an initially undirected graph, will be explained in detail in the Implementation section. We briefly describe the rooting concept here to explain the choice of this concept over Rafey's SCEFO actions.

The main idea behind rooting a linkage is to establish the parent/child relationship of the two links at a joint for each joint in the linkage. Thus, the rooting need only be done once for all joints, rather than specifying the child link each time a motion is desired. For example, in a human figure, a common root is the pelvis [ZHAO89]. This creates parent/child relationships between the links that account for most common transformations (i.e. the lower arm is rotated about the elbow, the leg is rotated about the hip, etc.) Once the rooting is done, the user can specify many transformations about joints which will respond correctly, and she does not need to specify which is the child link that will be transformed. This eliminates redundant input and basically allows the user to point to a joint and specify the transformation.

If the user then wants to transform about a joint in a manner that does not conform with the current rooting, she can re-root the linkage at the proper place and then specify the transformation. For instance, to rotate the upper leg about the knee, the user could root the linkage at either the foot or the lower leg, and the proper parent/child link relationships would be established.

In summary, the model proposed by Rafey had both strengths and weaknesses. The current model implemented by this project used his classifications of joints to reduce input, but it was necessary to create a new model of

the connectivity relationships expressed by joints in order to relieve the user of this burden.

3.2 University of Pennsylvania Jack System

As mentioned, researchers at the University of Pennsylvania have implemented the Jack system for manipulating jointed figures [BADL87], [ZHAO89]. This system has been developed from many years of research into both human figure animation and inverse kinematics solutions.

Jack allows the user to create scenes consisting of *segments*, *sites*, *joints*, and *constraints*. Segments are analogous to link objects, or physical objects in the scene. Sites are positions on segments at which joints can be placed to connect two segments. Joints are specifiable with every possible combination of degrees of freedom.

The specification of joints in this system is very flexible, and it allows a minimum of necessary input to describe a given joint. It has the drawback that two separate sites on two segments must be specified for each joint added to the scene. This is a result of the sites being specified relative to the coordinate frame of the segments. Ideally, a joint should be able to be specified by a single position/orientation, and the model will take care of the relative positioning of the links connected by the joint. An additional drawback is that any complex object in a scene must be built up by connecting segment objects with joints, even when the logical connection between these objects does not correspond to a joint. In SCEFO, the methods for building complex objects, such as *grouping* and *constructive solid geometry (CSG)* [CONN89], allow the user to specify a much wider range of complex objects in an intuitive fashion. The goal of the LINK package is to add to the methods for creating complex objects, rather than supersede existing useful methods.

The concept of a rooted linkage used in our model is taken from the model used by Jack. As described in [PHIL89], only the current child links are transformed when a joint transformation is requested. However, the implementation in Jack requires inversion of matrix transformations whenever the rooting of the linkage reverses the direction of the parent/child links. To avoid expensive inversions, the implementors of Jack keep two copies of the current transformation matrix at a joint, and simply use the appropriate matrix when calculating positions.

In our model, this redundancy is not necessary because the graph of the linkage itself does not depend on relative matrices for joints. Each joint maintains a global position and orientation that represents all transformations applied to it at the current time. As long as transformations are applied in the proper order to each object, the global position/orientation will always be correct.

The Jack system is also notable for its highly developed inverse kinematics motion control system. The system handles multiple reach goals in near to real time. Goals can be specified as positions, orientations, position and orientation, and several other types of goals such as aligning the end effector with a line in space, or orienting the goal so that it lies in a specified plane. Single or multiple goals can be specified for an articulated figure, with user-defined goal weights to determine goal priorities. The algorithm can be given a time limit if the current configuration of the linkage and goals is compute intensive. Since the algorithm is monotonically convergent toward at least one goal, it will simply be cut off at the given limit but will still have reached as far toward that goal as possible in the given time. In addition to inverse kinematics, direct manipulation of joints is available interactively. The pros and cons of the motion control system are discussed in Section 5.

Jack is a sophisticated system for creating and manipulating jointed figures, but it is very specific to this one purpose. While concepts from it have proven useful, incorporation of the overall structure would be difficult and impractical in BAGS.

3.3 MIT Media Laboratory

At the Computer Graphics Laboratory of MIT's Media Laboratory, a library of inverse kinematics routines exists for solving the reach problem. The algorithm uses the pseudo-inverse Jacobian [SIMS87] solution of minimizing joint angle changes to solve given reach goals. This technique produces a global perturbation of joint angles that tends to give a reasonably "natural" looking movement. Since all joint angles are typically updated at least slightly for a given reach, no one angle changes greatly unless that is necessary for the goal to be reached.

While this approach produces natural motion, its margin of error increases when the goal is further away. The implementation corrects for this internally by dividing the goal up into several subgoals which are points on a straight

line between the original position of the end effector and the end goal. The number of subdivisions is a parameter that can be changed, so the higher the number of iterations, the more likely the algorithm is to reach the goal.

One feature which was worth incorporating was that of preferred joint angles. In the Media Lab system, each joint is specified with a current angle and a preferred angle. As the joint is rotated and gets further away from the preferred angle, its motion in an inverse kinematics reach is dampened.

This allows more control for the animator over joints which might produce discontinuous or undesirable motion in a reach operation. This is implemented in the LINK package as a weighting parameter at each joint.

A disadvantage of this system that it has in common with Jack is that all joints are specified relative to some parent hierarchy. This makes it expensive for the direction of motion, or rooting of the linkage, to change, since recomputing all the transformation matrices is necessary. Another drawback is that there is no facility for direct manipulation of linkages; only inverse kinematics transformations can be specified.

3.4 Summary

The model adopted by this work for use in BAGS was influenced by the three systems described above, incorporating the notions of joint classification, a directed dependency graph for linkages, and preferred joint angles.

In addition to implementing these concepts, the LINK package and its underlying model also includes automatic interpolation for reach movements between specified frames; a concise interface to SCEFO for creating joint objects and applying transformations to them; and an entirely new inverse kinematics method for a fast solution to create animations via high-level motion control. The overall implementation of the LINK package is described in the next section.

4 Implementation

The implementation of this project consisted of two main elements.

1. Design of a SCEFO interface for creating and animating joints and linkages.

2. Creation of simulator software to process the modeling and animating requests for jointed objects. This software has three components:

   - Software to process the joint specifications in SCEFO.
   - Motion-control software to control direct manipulation (rotations and translations) of articulated objects.
   - Software implementing a new motion control method for 3-D inverse kinematics.

4.1 The Joint Model

The design of a model for joints took into account many factors. The previous research by Rafey showed that minimizing input when specifying joint parameters was important. The joints needed to be easily manipulable, since users would want to orient and position them correctly. There needed to be a simple facility for describing information about each joint. This information includes: joint type (BALL, PIN, etc); which one or two links the joint connects; joint limits on each of the appropriate degrees of freedom for the specified type; and other parameters possibly related to motion control methods, such as a weighting parameter for individual joints when applying an inverse kinematics algorithm.

In addition to modeling joints, users needed the ability to animate these joints and, correspondingly, the links attached to them. Two types of transformations were considered in this project.

- direct manipulation transformations. These consist of rotation or translation at a specific joint and degree of freedom within that joint.
- Inverse kinematics, which we will term a *reach* transformation.

For both these types of transformations, it is necessary for the user to indicate at which joint the transformation will take place (elbow, shoulder, etc.) To understand this, the concept of the *linkage dependency graph* and *chains* of joints and links is useful.

4.1.1 The Dependency Graph

A set of link objects connected by joints can be viewed as an undirected graph, where the links are the nodes of the graph and the joints are the edges (Figure 3). When the linkage is rooted, the graph becomes directed. The direction of all edges is determined by the root node. For each edge (joint) connected to the root node, the node at the other end of the edge becomes the child of the root node. In Figure 4, we show the example of the root node being the pelvis. Since the root node has edges connecting to the left thigh, right thigh, and torso, the pelvis becomes the parent to each of these nodes, and the edge between them takes on a direction. Once we have processed all the root node's edges, we apply the algorithm recursively to the child nodes of the root node. For each as yet undirected edge issuing from these nodes, the connecting node of that edge is made a child of the current node. Edges that have already been processed are marked so that the algorithm does not process them more than once. When the entire graph has been processed, we have a completely directed graph.

When a transformation at a joint is desired, the child node of the edge in the graph corresponding to that joint is determined. The subtree of the graph rooted at this node determines all link objects that will be affected by this transformation. If we transformed our human figure at the left hip joint, after rooting it at the pelvis, the subtree would consist of the left thigh node, left shin node, and left foot node, with the left hip, left knee and left ankle joints (edges) connecting these nodes (Figure 5.)

The simple subtree rooted at the child node of the input joint is sufficient for direct manipulation transformations such as rotate and translate. For joint reach transformations, we need to determine a *chain* of joints and links in the directed graph. This corresponds to the directed path from the input joint (edge) to an end effector link (node) specified by the user. The chain is determined by simple depth-first search through the subtree, and of course can not be determined unless the linkage has first been rooted. It is not necessary that the end effector node be a leaf of the graph; any node is legal as long as there is a path from the base joint to the end effector node.

The chain is necessary to indicate which edges (joints) can be used for manipulation during an inverse kinematics algorithm. For example, if the user specifies a joint reach transformation that begins at the pelvis joint and has the left hand as the end effector, the joints in the chain would consist of

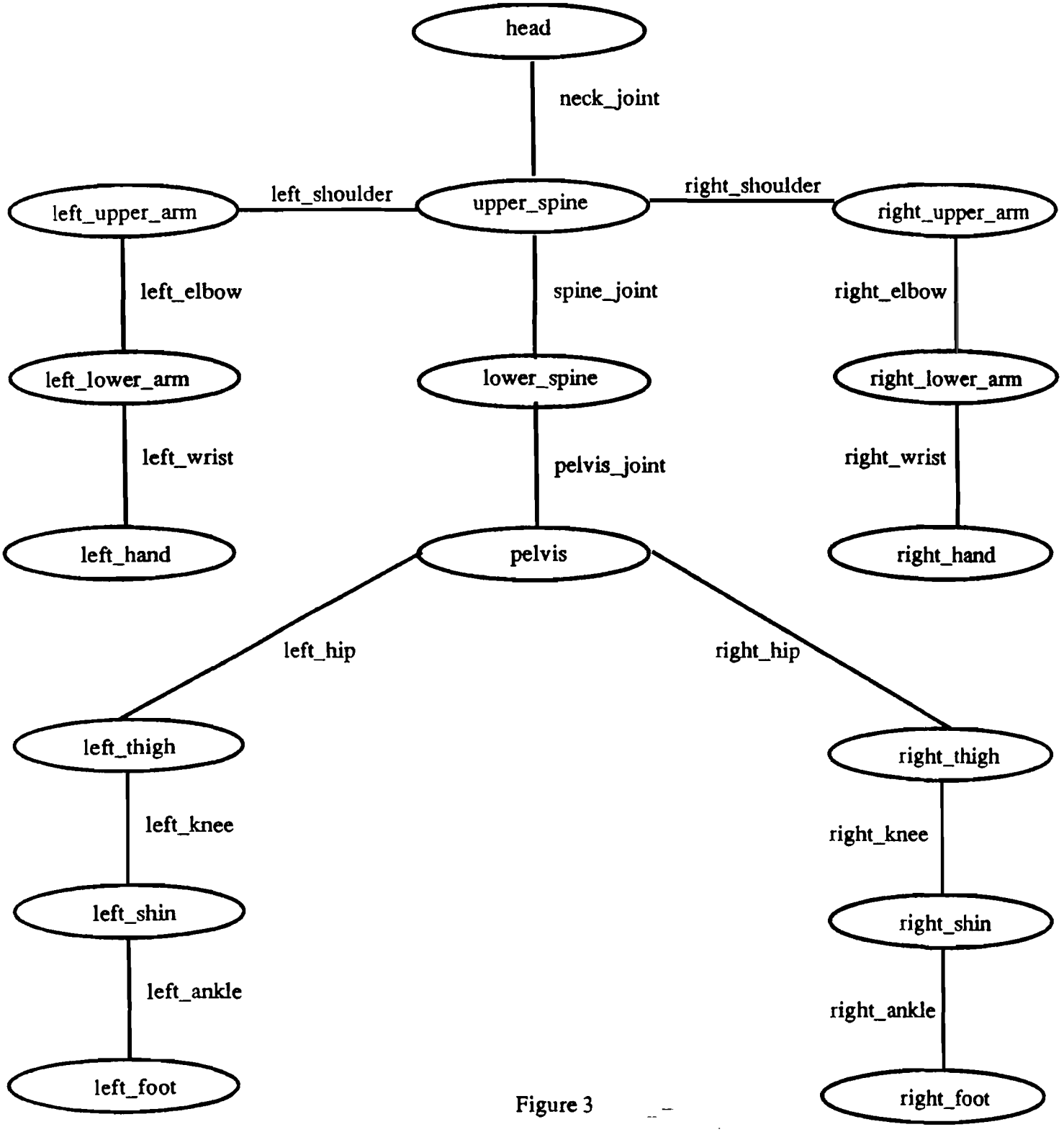


Figure 3

Undirected graph

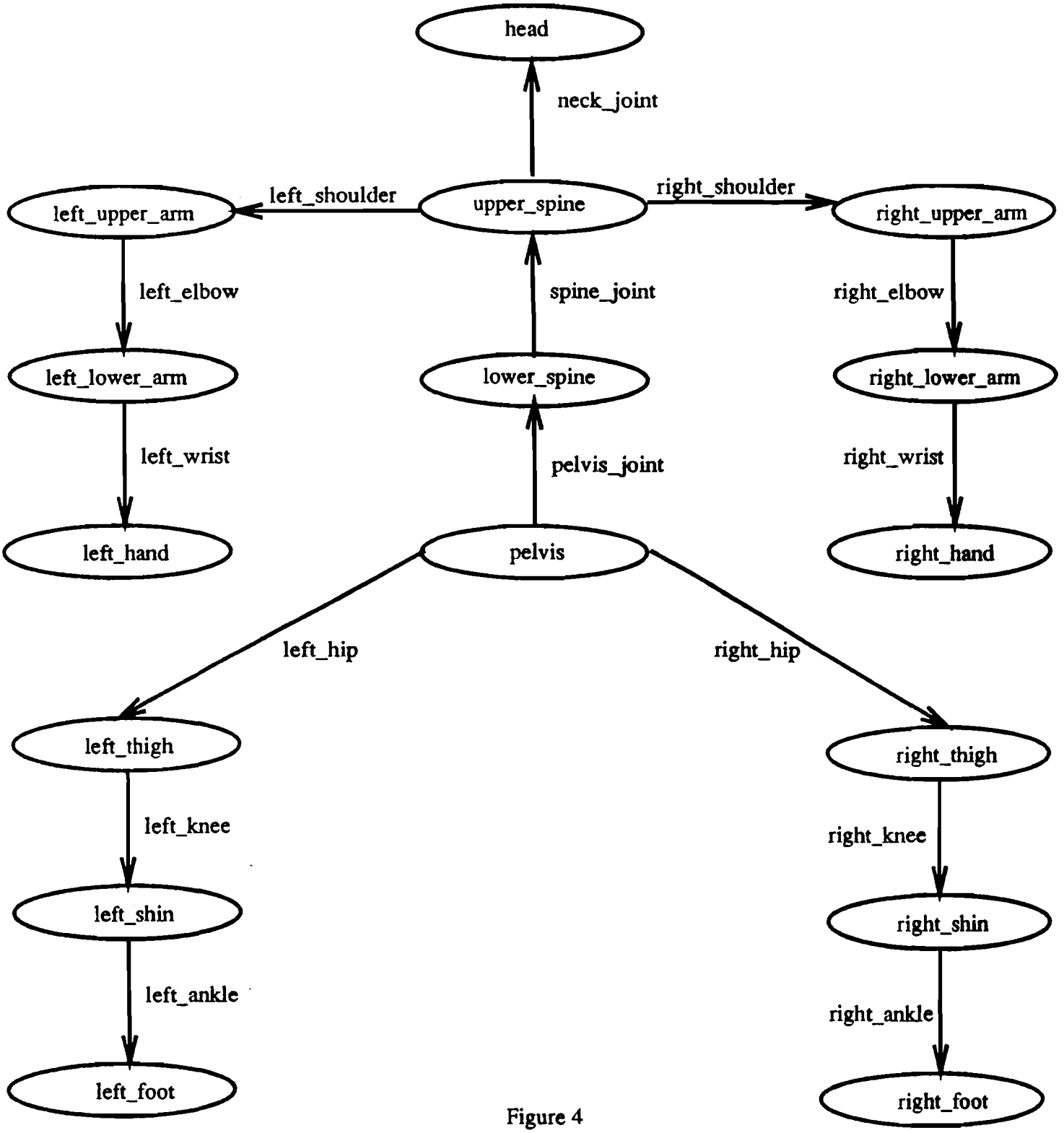


Figure 4

Directed graph with root at pelvis

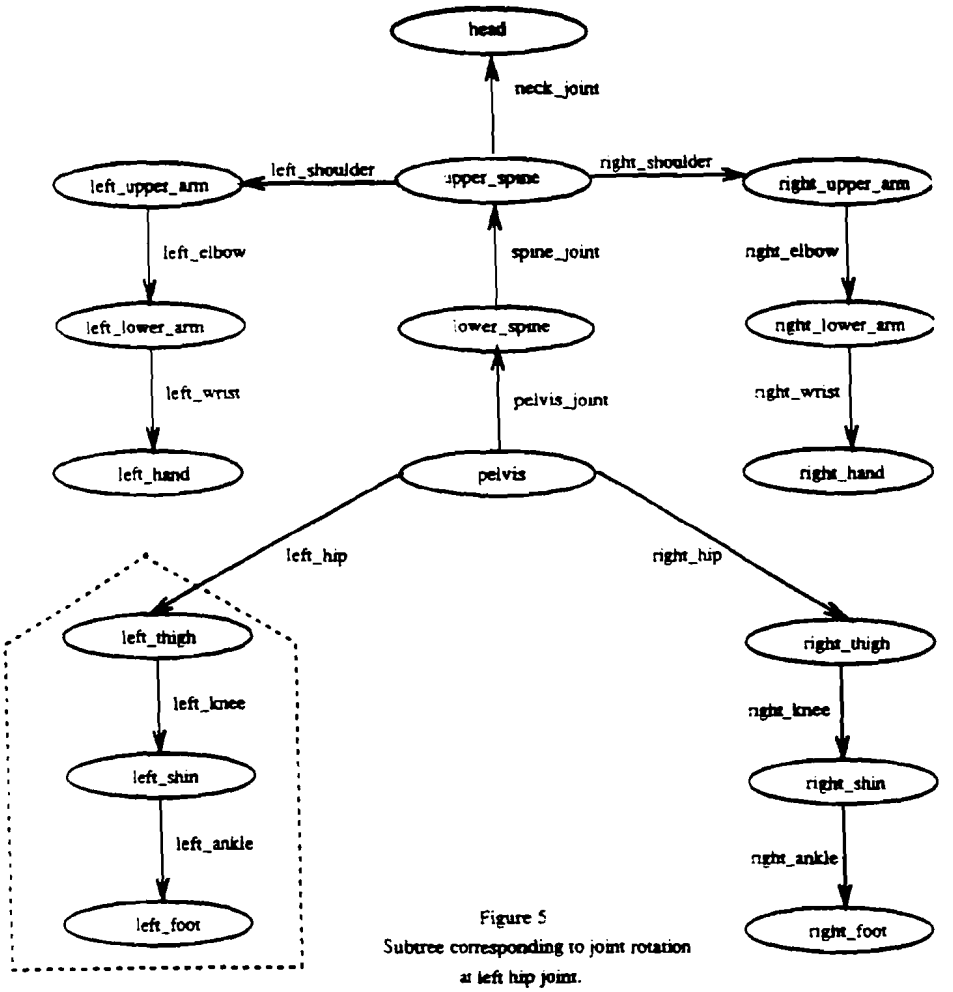


Figure 5
Subtree corresponding to joint rotation at left hip joint.

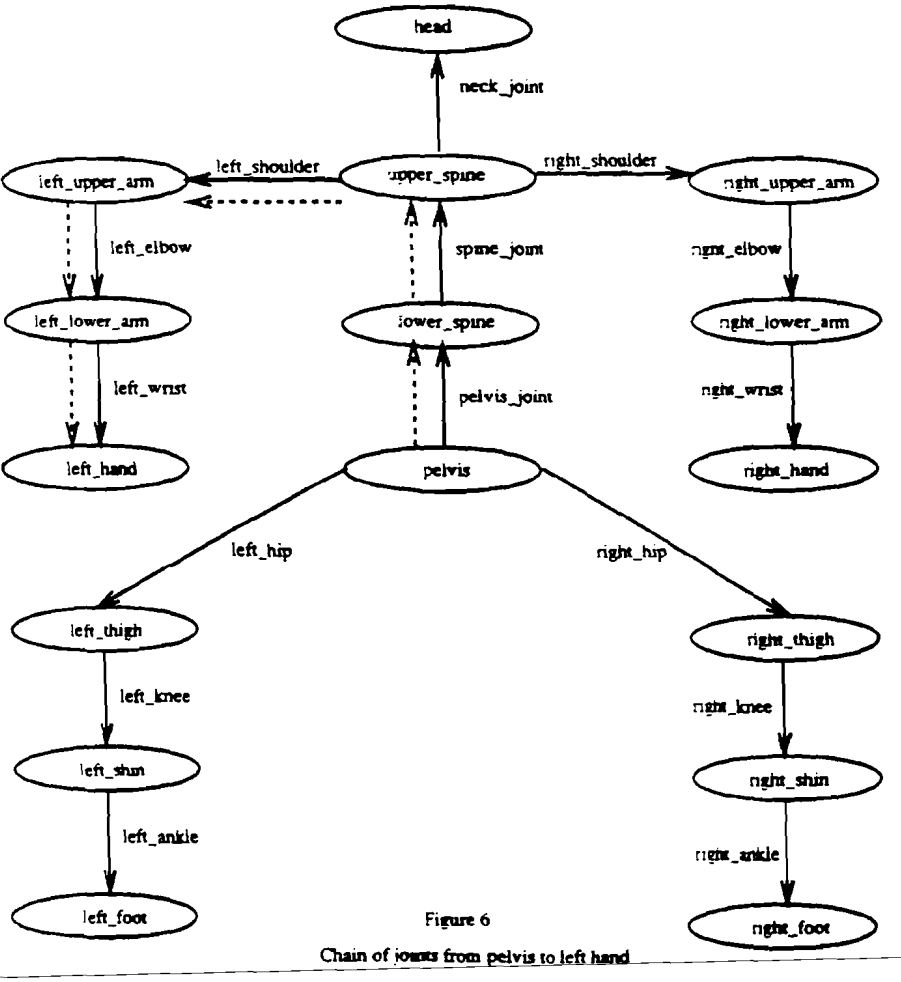


Figure 6
Chain of joints from pelvis to left hand

the pelvis joint, spine joint, left shoulder, left elbow, and left wrist (Figure 6.) Of course, more links than just the child links of these joints may be affected by the movement. For instance, the head node will follow along if the figure bends at the pelvis joint. However, the neck joint will not be used during the inverse kinematics calculation because there is no way it would help the left hand achieve its goal. Correspondingly, it is not along the direct path from the pelvis node to the left hand node in the graph.

Once the subtree that is dependent on the joint transformation is determined, the transformation can be applied to all dependent nodes in the subtree. If the transformation is of the reach variety, the chain is built and its member joints can be used to attempt to reach the goal.

The determination of the dependency graph, subtrees and chains is transparent to the user, although she must be familiar with the concept of rooting the linkage. The main user interface issue lies in the fact that almost all input can be regarded as changes to the joint. For modeling, parameters need to be set that indicate connectivity, joint type, etc. Animation transforms can also be viewed as changes that happen at the joint, and affect all dependent links.

Since the input necessary for both modeling joints and animating them relates directly to the joint itself, the choice was made to make the joint a SCEFO object. SCEFO objects have the property that they can take parameters or fields as input to describe characteristics of the object. In addition, all regular transformations that can be specified via SCEFO change operations (*change ops*), can also be applied to joints if they are modeled as objects. They can be rotated, translated, etc., and can thus be easily positioned and oriented by the user. Modeling the joint as a SCEFO object allows a simple and uniform interface to the user for specifying all information about a joint.

In the past year, the concept of *meta-objects* in BAGS has been developed. Typically, meta-objects have no visibility in the scene, but contain information that is necessary for the modeling or animation of other complex objects. The point and path objects are examples of this. A typical usage of a set of point objects is to designate control points in a spline. A path object, often determined by the positions of point objects, can also be used for such purposes as defining the contour in an object of revolution.

Since the point meta-object already existed, it seemed the logical choice as the model for holding information about joints. Accordingly, a new mem-

ber of the point object class, called the joint, was added. In order for users to model joints in their scene, they simply need to create templates of joint objects, and set all necessary fields to define the characteristics of the joint. The creation of templates and setting of fields, along with other SCEFO particulars, is described in [CONN89]. The following is an example of modeling a shoulder joint:

```
template (left_shoulder) joint;

change (left_shoulder)
   set_field <0, {"joint_type", BALL}>,
   set_field <0, {"link1", "torso"}>,
   set_field <0, {"link2", "left_upper_arm"}>;
```

Modeling field values in the joint object have the following meaning:

- *joint_type* – type of joint such as PIN joint, BALL joint, etc. Types correspond to the sixteen classified by Rafey and Beyer.
- *link1, link2* – two links that the joint is connected to. Must be the names of SCEFO objects in the scene.
- *limit_rot_x, limit_rot_y, limit_rot_z,*
  limit_trans_x, limit_trans_y, limit_trans_z
  – several parameters which can be used to set limits on the appropriate degrees of freedom for the specified joint type.
- *weight* – weight to give this joint when invoking the inverse kinematics algorithm. This can be used to dampen the motion at some joints in order to create a more natural animation.

The joint object now contains the information of connectivity between the torso and the left upper arm, and the type of joint that exists at this connection.

To animate about a joint, the interface is uniform – additional set_field change ops are added to the left_shoulder joint object:

```
change (left_shoulder)
   set_field <1, {"joint_rotate", {{0.0, 1.0, 0.0}, 30}}>,
   set_field <2, {"joint_reach", {{goal:_position}, "left_hand"}}>;
```

This fragment creates two transformations to the joint and its dependent link objects. At time 1, the joint will be rotated 30 degrees about its object space Y axis. At time 2, the chain starting at the left shoulder and terminating at the left hand will attempt to reach for the point in space indicated by the dependency *goal:_position*. This dependency represents the point in space occupied by the center point of the object named ***goal***, and will be evaluated at time 2 to determine the exact coordinates of the point.

Although all connectivity information is contained in the joint objects via the "link1" and "link2" parameters, there is no mechanism with just joint objects to partition joints into separate linkages. Initially, separate linkages were all treated as one large linkage. This was feasible, since the graph of the linkage would simply have several unconnected components. However, treating each linkage as an individual entity has the advantage that users can selectively update certain linkages without having computation performed on every joint object in the scene. This partitioning speeds up processing time, if there are many linkages, and makes the logical distinction between each linkage conceptually clearer when modeling. The addition of a linkage object also allows change ops to be applied to it that are more naturally associated with an entire linkage, such as setting the root.

Accordingly, the user must also create a "linkage" object in the SCEFO script. This is of type linkage, another meta-object member of the point object class.

The set_field change ops currently implemented for linkage objects are:

1. *add_joint* – adds the named joint object to the linkage

2. *root* – roots the linkage at the named link object

The following is an example of SCEFO to create a linkage:

```
template (fred) linkage;

change (fred)
   set_field <0, {"add_joint", "left_knee"}>,
```

```
set_field <0, {"add_joint", "left_hip"}>,
set_field <0, {"add_joint", "left_ankle"}>
set_field <0, {"root", "pelvis"}>;
```

This SCEFO fragment creates a leg linkage which is rooted at a stationary link object, the pelvis. Although only the two set_field change ops listed have been implemented so far, additional change ops for linkages, such as "*delete_joint*", are envisioned. Ideally, the simulator code would be able to handle any normal SCEFO change op to a linkage, and would then update the individual components of the linkage correctly. This would allow the user to do things like rotate, translate or scale the entire linkage in world space.

For a complete SCEFO script modeling a simple human figure, see Appendix A. For detailed usage when creating joints and linkages, see [BORD90].

4.2 Simulator Software

The simulator software is currently contained entirely in the LINK package [BORD90]. The LINK package is a simulator server which can be called by such clients as the interactive animator (MOO), and renderers. Currently, the LINK package is being used by MOO, although renderers have not yet been integrated as clients. Issues surrounding simulator interaction in a single scene are still in the research phase.

A description of the LINK package's interaction with MOO will suffice to describe its main features. When MOO detects that a linkage has been added to the scene, it calls the *LINKactivate()* procedure with a linkage name and time. At that time, the LINK package creates a dependency graph for the current linkage, and adds all transformations to joint objects contained in the linkage to its own internal list. Once it has read all transformations, it applies the transformations that affect the current time to the appropriate objects, and updates the representation of those objects via DBOSS.

Since the simulator is aware of all transformations to a linkage once it is activated, it only needs to be notified in two cases:

1. If a joint object in the linkage has a change op added or modified; and

2. If the current time changes.

In case 1, the simulator reevaluates the modified change op and updates its own internal representation. If it is a new change op, it is added to

the internal list of transformations. Notification of modified or new change ops is handled by DBOSS, which allows clients such as the simulator to express interest in changes to objects, and receive update messages when these changes occur.

In case 2, the transformations appropriate for the current time are applied so that the simulator correctly updates all objects it is controlling. This notification must come directly from the client of the LINK package (in this case MOO), since DBOSS has no notion of the current time.

This simple interface to the LINK package has made it fairly straightforward to model and animate linkages in MOO. Although not fully implemented yet, the user interface in MOO for linkages will be simple to complete and will provide a fully interactive environment for both creation of complex jointed objects, and high-level specification of motion via direct manipulation and inverse kinematics. The SCEFO basis exists, and it is a simple matter of translating user input choices into SCEFO statements. The interactive nature of DBOSS takes care of the coordination between MOO and the LINK package.

4.3 Motion Control in the Simulator

There are currently three motion change ops that can be applied to a joint object: *joint_translate*, *joint_rotate*, and *joint_reach*. The simulator code is responsible for generating the motion of dependent link objects when these change ops are added to the scene. It is also responsible for correctly interpolating these change ops when they contain more than one control point. The following sections examine the two motion control methods of direct manipulation and inverse kinematics, as well as the interpolation issue.

It should be noted that both motion-control methods maintain joint limit constraints as well as connectivity constraints. If a transformation attempts to rotate or translate a joint past its limit, a warning message is printed and the joint is only transformed as far as the limit allows.

4.3.1 Direct Manipulation

Direct manipulation of a joint and its dependent links is relatively easy once the linkage has been rooted. Since the graph representing the linkage is fully directed after rooting, the subtree rooted at the child node of the joint will

represent the links that need to be transformed. When a direct manipulation such as *joint_rotate* is requested, an identical rotate is applied to all the proper link objects, and their positions and orientations are updated accordingly. To do this, a simple depth-first traversal of this subtree is performed.

4.3.2 Inverse Kinematics

As mentioned previously, the input to any inverse kinematics method needs both a base joint for motion, an end effector link object, and a goal. The end effector represents the link that is "reaching" for the goal. Optionally, the input can also include a "site" on the end effector to direct the reach a little more specifically. An example of this refinement would be the specification of the hand as the end effector, and the specification of a fingertip as the actual site that makes contact with the goal.

When the simulator code processes a *joint_reach* transformation, it first builds the chain of joints that will be used to try and reach the goal. This chain, along with the goal and end effector site, are sent to the main inverse kinematics module. Since any inverse kinematics method will need exactly this input to compute a solution, new methods can be easily added.

The current method implemented in the LINK package is an iterative algorithm inspired by an idea first implemented in 2-D. Some inverse kinematics algorithms rely on expensive matrix inversion techniques (see Section 5.) This new method iterates through the chain of joints one by one, using the local information to determine how closely this joint can help the end effector reach the goal.

At each joint in the chain, the algorithm projects the goal point into the plane of movement defined by the end effector and its axis of transformation, which is determined by the current joint. If the joint is a translational joint, the goal point is projected onto a line through space which represents the allowable motion of the end effector relative to this joint. Once the goal is projected, a change to the joint angle or translational value is computed which will bring the end effector as close as possible to the goal, and this transformation is applied for the current joint and all its dependents.

The iteration through the chain starts at the most distal (that is, furthest from the base of the chain) joint and iterates backwards through the chain to the base joint. If we use a human arm as an example, this means that the algorithm first tries to reach the hand toward the goal, then the lower

arm, and finally the upper arm which is controlled by the base joint, the shoulder. If each of these iterations were viewed separately, the motion would look awkward and unnatural. However, all iterations are concatenated into a single movement, and only the final position is actually updated in the scene.

Figure 7a shows a simple illustration of a 2 dimensional arm-like linkage in an intitial configuration. Figure 7b shows the linkage in several intermediate configurations after a series of iterations through the chain. Figure 7c gives the final configuration of the arm as it reaches as closely as possible to the goal.

4.4 Interpolation

An important issue that the simulator code must deal with is that of interpolation of motion. For each time in a set of control points specifying a joint transformation, the simulator outputs a global position/orientation via the SCEFO *orient* change op. This is done for each object in the linkage, links as well as joints. This takes care of the control point times, but the simulator may then be requested by a client program, such as MOO, to produce position/orientation for the objects at interpolated times between the two control points.

The values of an *orient* change op include the *from, at* and *up* points of the object – that is, the position of the object and its orientation. Since this information may be the result of several transformations (i.e a *joint_translate*, *joint_rotate*, or *joint_reach*, which may itself contain many rotates and translates), there is no way to use standard interpolation techniques between two global *orient* operators. There is no retention of the motion that produced the orient, and a standard interpolation technique, such as linear, is not guaranteed to produce correct results.

Accordingly, the simulator must somehow generate these global in-between positions itself. For a simple transformation such as a *joint_rotate*, this is not difficult. Consider the following example:

```
change (left_shoulder)
  set_field <0, {"joint_rotate", {{1.0, 0.0, 0.0}, 0}}>
            <8, {"joint_rotate", {{1.0, 0.0, 0.0}, 90}}>;
```

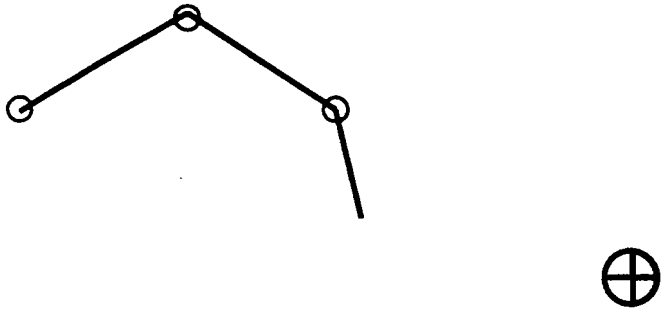


Figure 7a

Initial configuration of arm and goal

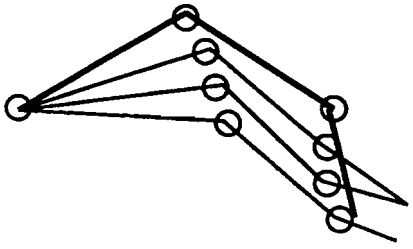


Figure 7b

Intermediate arm positions after several chain iterations

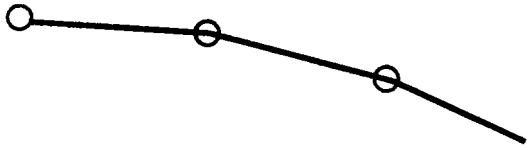


Figure 7c

Final configuration of arm reaching as closely as possible to goal

The intention of this change op is to rotate the left arm at the shoulder by 90 degrees, between times 0 and 8. The simulator code generates internal transformation structures that indicate the shoulder should be rotated by 0 degrees at time 0 and 90 degrees at time 8. If it is then requested to produce positions for the objects in the arm at time 4, it generates a transformation that rotates the shoulder by 45 degrees. However, internally it must remember that at time 4 it rotated by 45 degrees, so that at time 8 it now only needs to rotate by another 45 degrees, rather than the full 90. Results of transformations are concatenated in script/time order and are cumulative.

Keeping track of what has happened previously (i.e. what in-between control points have been generated) is straightforward for simple transformations such as *joint_rotate* and *joint_translate*. However, in a *joint_reach* transformation, the same technique cannot be applied. There is no way for the simulator to look back in time and subtract out the motion that has already been computed, because this motion was produced by a series of rotations/translations generated by the inverse kinematics algorithm, and these specific transformations are not saved. All the simulator will have knowledge of is the position and orientation of the arm after the last control point value was applied.

Interpolation of *joint_reach* transformations is handled by interpolating the goal point, using the interpolation method specified for the control point containing the reach transformation. Discrete reach transformations to each of the interpolated goals are then generated. In the following SCEFO fragment, the user wants the arm to reach from point {1.0, 1.0, 1.0} to the position of a stationary ball, between times 0 and 8:

```
change (left_shoulder)
   set_field <0, {"joint_reach", {{1.0, 1.0, 1.0}, "hand"}}>
             <8,{ "joint_reach", {ball:_position, "hand"}}>;
```

If the client of the simulator then requests output for time 4, the simulator will generate an intermediate goal point that is halfway between points {1.0, 1.0, 1.0} and the *_position* of the ball. This will result in the arm reaching for {1.0, 1.0, 1.0} at time 0, the intermediate goal at time 4, and the ball's *_position* at time 8.

This approach produces reasonable motion since it is based on the interpolation method specified by the user in the control point. The method

defaults to linear interpolation if none is specified, which means the linkage will move as much as possible in a straight line toward the goal. However, other forms of interpolation such as spline or discrete can also be specified.

In addition to generating interpolated motion between control point times, the simulator generates extra motion if a joint transformation is dependent on an object that is moving. If the user specifies the following SCEFO transformation:

```
change (leftshoulder)
   set_field <0, {"joint_reach", {ball:_position, "hand"}}>;
```

the arm will reach for the ball at time 0. If the ball ever moves, at time 0 or otherwise, additional transformations are generated to keep the arm following the ball. This maintenance of dependencies, which is implemented on top of the SCENE Correspondences and Dependencies package (SCAD) [GOLD89], allows complex animation to be produced with a single transformation.

One problem that *joint_reach* transformations produce is that of granularity. The intermediate goal positions and corresponding reach transformations are only generated if the the client program, such as MOO, requests positions of objects at intermediate times. If the granularity is high, the goals will be more widely spaced. Reaching for a goal that is far away will generally produce a different motion than reaching for a goal that is close. Thus a user is not guaranteed to get the same motion each time the simulation is run, unless the exact same sequence of frames is requested.

This is not really a problem in an interactive setting, where typically all the frames are regenerated each time an animation is played back. However, if the simulator is integrated into a rendering program, frames may be generated in a random sequence and sometimes with varying granularity. It might be wise to enforce some specified granularity in this setting, so that users do not unknowingly get incompatible results from frame to frame.

4.5 Summary

The implementation described in this section produced a new SCEFO interface for creating and animating articulated objects in BAGS. This implementation allows for multiple linkages to be created and maintained over

time, with both direct manipulation of the joints in the linkage, and indirect manipulation via the high-level inverse kinematics algorithm. In addition to providing this tool for creating and manipulating a new class of complex objects, it was one of the first simulator methods implemented on top of DBOSS and proved to be a valuable test of that package, its functionality and reliability.

5 Comparison of Inverse Kinematics Algorithms

The selection of an inverse kinematics algorithm for high-level motion control was based on the need for a real time algorithm which produced reasonably natural motion. A brief comparison of the 3 methods considered, in terms of efficiency, visual effectiveness, and robustness will be useful to explain the choice of method for this project. Each of the methods in the following list has been described in the Related Work and Implementation sections.

1. Pseudo-inverse Jacobian solution used in MIT Media Lab robotlib.
2. Multiple constraints solution used by Jack system at the University of Pennsylvania.
3. Iterative linear solution developed at Brown for this project.

5.1 Efficiency

In choosing the algorithm implemented by this project, computational complexity was a key factor in its desirability over the other methods. If we consider the number of joints in an inverse kinematics chain n, we can evaluate the relative complexity of the methods. Solution 1 computes the inverse of a matrix of size $n \times 3$ or $n \times 6$, depending on whether the goal includes orientation information. A normal Gaussian elimination solution for inverting the matrix is an $O(n^3)$ operation. Solution 2 also relies on a matrix inversion and is $O(n^2)$ [ZHAO89]. In contrast, solution 3 does a simple iteration through the chain using the local information at each joint to determine the delta joint angles. A constant amount of work is done at each joint. While

the iteration through the chain may be performed a number of times before the end effector converges to the goal, there is a maximum number of iterations which keeps the algorithm linear, or $O(n)$.

5.2 Visual effectiveness

Each of the three methods can be demonstrated to produce reasonably natural motion for specific configurations. The motion produced by solution 1 was the least tested of the three methods, mainly due to lack of robustness in the implementation. However, a superficial assessment showed that the global nature of the pseudo-inverse method produced reasonable effects, with the change in joint angles distributed evenly throughout the chain. Although this motion will not always be desirable (for instance, there is no need to rotate at a shoulder joint if a goal is reachable simply by raising the forearm), in other cases the global motion mimics what happens in reality.

Solution 2 produced very nice motion in near to real time. Since the implementation uses a technique of subdividing the goal up into a series of less distant subgoals, the motion monotonically converged toward the goal as the iterations progressed. In addition, the ability to specify multiple prioritized goal made this a very attractive system for producing complex animations. Unfortunately, the complexity of the system also worked against its inclusion into the LINK package at this time.

Tests of solution 3 in random configurations determined that reasonable motion can be produced with proper use of joint limits and weights. A completely unconstrained linkage using this method will tend to flop around, always bending as far as possible at each joint in order to reach the goal. This would, for instance, cause a human figure to possibly bend over backwards to reach a troublesome goal, or to bend an excessive amount at the pelvis when a less drastic change in all its joint angles would be more natural. However, imposing joint limits eliminates the physically impossible joint angle changes; and imposing joint weights tended to eliminate the excessive bending problem. Future research into improving this algorithm would seem likely to improve its visual effectiveness. For example, an occasional reversal of the order in which the joints are traversed might produce less abrupt motion.

5.3 Robustness

The issue of robustness of the implementation was a serious concern in solution 1. The inversion of the pseudo-Jacobian matrix was not always guaranteed to succeed, which leads to anomalies in the motion if many reaches are being calculated sequentially. Additionally, the error margin for the calculations tended to increase dramatically when the goal was further away. The solution also produced erroneous results when the reaching linkage was in a "singular" configuration [15], such as being extended straight out.

There was no opportunity to evaluate solution 2 in terms of robustness, other than superficial assessment of the live demo. In that demo, the goals were nearly always achieved and no errors occurred. However, the linkage did occasionally get caught in a "kinked" position where a joint would no longer move even though by doing so the goal might be achieved.

Solution 3 was similar to solution 2 in that most goals that were reachable were in fact reached using this algorithm. Since the calculation at each joint is a simple check for angle or translational distance between the end effector and the goal, matrix inversion errors are not a factor. Although no formal calculation of the workspace reachable by a given linkage was done, such as is proposed in [KORE84], the repeated usage of the method on several models confirmed its general robustness.

Some configurations did produce the same "kinking" effect that was mentioned for solution 2. This is an artifact of the locality of information used when determining joint angle changes. It is interesting to note that the more expensive method and this new method produced a similar result.

5.4 Summary

In summary, the simple iterative inverse kinematics method developed at Brown was chosen for inclusion in this project on the basis of its efficiency and ability to produce reasonable motion. The Media Lab system was too error prone and computationally expensive to be useful at this time. The University of Pennsylvania code is a highly developed system and would very likely produce good animations. However, its complexity was a discouraging factor when attempting to integrate the code into BAGS. Since the visual results did not vary significantly, the simpler, more efficient $O(n)$ algorithm

was the most practical choice. In the future, it would be interesting to integrate both of these outside methods and do a direct comparison of the three algorithms under identical conditions.

6 Issues for Future Research

The work described in this paper provided an initial interactive environment for creating and animating linkages. Motion control methods were provided for direct manipulation of linkage components, and for high-level inverse kinematics manipulation.

While this basis provided a platform for creating a new class of complex objects, and easily specifying animations of these objects, much work remains to be done. Experimentation with the system and the actual implementation revealed several issues that will require further research.

A key component missing in the current implementation is the ability to dynamically add and delete joints from a linkage, or dynamically alter the characteristics of a joint during the course of an animation. This would allow interesting effects to be created such as the gradual restriction of freedom at a joint, the connection of two linkages (one can envision two robots clasping hands dynamically and having the first robot drag the second along), or the breaking off of a subcomponent into an independent linkage.

The motion produced by the inverse kinematics simulator also provides a basis for much further research. While reasonably realistic motion is produced using the constraint mechanisms of joint limits and weights, a truly realistic simulation would need to take into account the physics of motion, and use techniques such as *dynamic analysis* [BARR88], or *physically-based modeling* [BARR89].

Analysis of the simulator motion also clearly shows the need for a collision detection simulator of some kind. Again, joint limits will help but they do not realistically model collision detection. A collision detection simulator would have to work in concert with the joint motion simulator, which raises the issue of simulator interaction, a rich area for further research.

The inverse kinematics algorithm implemented in this project can be improved by research into several areas:

- A more sophisticated weighting algorithm would produce better damped motion of the joints. The current algorithm is a simple linear weighting

that sometimes prevents a goal from being reached. In addition, a fast interactive previewer of simple animations produced by assigning different weights would give the animator essential assistance in determining the correct values.

- Solution of closed-cycle linkages should be added. This involves finding solutions for joint chains that have multiple paths from the base joint to the end effector [MOTT88]. Currently, cyclical configurations of joints and links are allowed, but only one path from base joint to end effector is used. The other paths are ignored. Adding cycles also introduces ambiguity to the rooting algorithm, and this would need to be resolved. A fast algorithm for closed cycles would be a significant contribution to the research in this area.

- A scheme for solutions of multiple goals is also a useful research aim. While the current system will solve multiple goals, there is no provision for joint chains that overlap. One goal will override another in an arbitrary fashion. When multiple conflicting goals exist, there needs to be some sort of prioritization scheme to determine which goal, if any, should be solved.

7 Conclusion

The project described in this paper provided an extension of the BAGS environment to allow modeling and animation of a new class of complex object, the articulated object. A basic interface to both the animation scripting language, SCEFO, and the interactive animation environment was provided.

A new real time solution to the inverse kinematics problem was implemented, providing a high-level motion control method for animating linkages. In the first pass at implementing the algorithm, the goals of reasonable performance, reasonably "natural" motion, and the ability to create task oriented animations were realized.

In conclusion, the addition of articulated objects and associated simulation methods to the BAGS environment is a contribution to both interactive modeling and interactive animation. It will provide a basis for future development of complex animations, as well as a research testbed for motion control and simulation methods.

8 Appendix A – Example SCEFO Script

```
/* Part A & B:  include/read the necessary files */
#include <joints.scefo>

read ("cube.off") cube;
read ("joint.off") cube;
read ("linkage.off") linkage;
read ("sphere.off") sphere;

/* Part C:  model the links */
/* These are the links */
template (limb) cube;
change (limb) scale <0, {0.5,0.05,0.05}>,
              rotate <0, {{1.0, 0.0, 0.0}, 45.0}>;
template (leftupperarm) limb;
change (leftupperarm) rotate <0, {{0.0, 0.0, 1.0}, -90.0}>,
                      translate <0, {0.7, 1.5, 0.0}>;

template (leftlowerarm) limb;
change (leftlowerarm) rotate <0, {{0.0, 0.0, 1.0}, 90.0}>,
                      translate <0, {0.7, 0.5, 0.0}>;
template (hand) cube;
change (hand) scale <0, {0.1, 0.15, 0.01}>,
              rotate <0, {{0.0, 1.0, 0.0}, 45.0}>;
template (lefthand) hand;
change (lefthand) translate <0, {0.7, 0.0, 0.0}>;

template (torso) limb;
change (torso) rotate <0, {{0.0, 0.0, 1.0}, 90.0}>,
               translate <0, {1.0, 1.5, 0.0}>;

/* Part D:  model the joints */
template (jointleftshoulder) joint;
change (jointleftshoulder)
                    set_field <0, {"link1","torso"}>,
```

```
                    set_field <0, {"link2","leftshoulder"}>,
                    set_field <0, {"joint_type",BALL}>,
                    rotate <0, {{1.0, 0.0, 0.0}, 90.0}>,
                    translate  <0, {0.7, 2.0, 0.0}>;

template (leftelbow) joint;
change (leftelbow) set_field <0, {"link1","leftupperarm"}>,
                   set_field <0, {"link2","leftlowerarm"}>,
                   set_field <0, {"limit_rot_x", {-180.0, 180.0}}>,
                   set_field <0, {"joint_type",PIN}>;

template (leftwrist) joint;
change (leftwrist) set_field <0, {"link1","leftlowerarm"}>,
                   set_field <0, {"link2","lefthand"}>,
                   set_field <0, {"limit_rot_x", {0.0, 180.0}}>,
                   set_field <0, {"joint_type",PIN}>,
                   translate  <0, {0.7, 0.0, 0.0}>;


/* Now group the links and joints so you can move them altogether */
/* This movement must come before the linkage object template */
group (body) torso, leftupperarm, leftlowerarm, lefthand;
group (joints) jointleftshoulder, leftelbow, leftwrist;
group (both) body, joints;

change (both) translate <0, {-4.0, -2.0, 0.0}>;

/* Part E: */
/* Now add the linkage object */

template (arm_linkage) linkage;

change (arm_linkage) set_field <0, {"root",      "torso"   }>,
                     set_field <0, {"add_joint", "jointleftshoulder"}>,
                     set_field <0, {"add_joint", "leftelbow"}>,
                     set_field <0, {"add_joint", "leftwrist"}>;
```

```
/* Part F: */
/* Now add any joint movement set fields that you want */

template (ball) sphere;
change  (ball) translate <0, {-2.0, 1.0, -0.3}>;

change (jointleftshoulder)
    set_field <0, {"joint_reach", {{0.7, 0.0, 0.0 },"lefthand"}}>
              <5, {"joint_reach",  {ball:_position}, "lefthand"}}>;
```

References

[BADL87] Badler, Norman I., Manoochehri, Kamran H., and Walters, Graham, *Articulated Figure Positioning by Multiple Constraints,* IEEE Computer Graphics and Applications, June 1987

[BARR88] Barr, A. H., and Barzel, R., *A Modeling System Based on Dynamic Constraints,* ACM SIGGRAPH Proceedings, 1988

[BARR89] Barr, Alan H., *Introduction to Physically-Based Modeling,* Course Notes #30, Topics in Physically-Based Modeling, ACM SIGGRAPH '89, Boston, MA

[BEYE88] Beyer, Thomas B., *Dynamic Simulation and Dynamic Constraints within an Interactive Animation System,* Masters Thesis (Draft), Brown University, Dept. of Computer Science, August 1989

[BORD90] Borden, Lisa K., *A Guide to Linkages,* Brown University Computer Graphics Group, Providence, RI, March 1990

[CONN89] Conner, D. Brookshire, *A Guide to New Scefo,* Brown University Computer Graphics Group, Providence, RI, August 1989

[FERD86] Ferdman, Alejandro Jose, *Robotics Techniques for Controlling Computer Animated Figures,* Masters Thesis, Massachusetts Institute of Technology, Dept. of Architecture, 1986

[GOLD89] Fiske, Barton C., and Gold, Melissa Y., *Overview of BAGS Internals for Packages and Commands,* Brown University Computer Graphics Group, Providence, RI, August 1989

[ISAA87] Isaacs, Paul M. and Cohen, Michael F., *Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Inverse Dynamics,* Computer Graphics, Volume 21, Number 4, July 1987

[JASM89] Jasmin, Pierre, *Introduction to Computer Animation,* Course Notes #9, Introduction to Computer Animation, ACM SIGGRAPH '89, Boston, MA

[KORE84] Korein, James U., *A Geometric Investigation of Reach,* MIT Press, Cambridge, 1984

[MOTT88] Mott, David C., *An Animation System for Motion Specification of Articulated Rigid Bodies in Scientific Visualization,* Masters Thesis, Brown University, Dept. of Computer Science, May 1988

[PHIL89] Phillips, Cary B., *Jack User's Guide,* Computer Graphics Research Laboratory, University of Pennsylvania, October 1989

[RAFE88] Rafey, Richter A., *A Model for Interactive Specification and Animation of Jointed Objects,* Masters Thesis, Brown University, Dept. of Computer Science, May 1988

[SIMS87] Sims, Karl, *Locomotion of Jointed Figures over Complex Terrain,* Masters Thesis, Massachusetts Institute of Technology, Dept. of Architecture, 1987

[STRA88] Strauss, Paul S., *BAGS: Brown Animation Generation System,* Technical Report CS-88-22, Brown University, Dept. of Computer Science, May 1988

[ZELT85] Zeltzer, David, *Toward an Integrated View of Computer Animation,* Proceedings of Graphics Interface 1985, Montreal

[ZHAO89] Zhao, Jianmen, and Badler, Norman I., *Real Time Inverse Kinematics with Joint Limits and Spatial Constraints,* University of Pennsylvania Technical Report MS-CIS-89-09, December 1989