Computer Animation for Scientific Visualization

Michael E. Bauer & Ken C. Hu
Sterling Software, Palo Alto, CA, U.S.A.

Abstract

During the past few years computer animation has proven itself as a powerful tool to assist researchers in understanding complex scientific data sets. One major scientific application of computer animation is the field of computational fluid dynamics. In this field, animating the time dependent numerical solutions of such phenomena as shock waves, vortices, shear layers, and wakes is of major importance.

This paper concentrates on the mathematical representation of transformations for hierarchical objects such as translation, rotation, and scaling which is essential for animation. A description of the application of these transformations between world and local coordinate systems is presented. Several interpolation techniques to define the motion in key frame based animation systems are discussed. Comprehensive C-code covering these topics is provided in appendix B.

A brief description of other animation techniques such as cycled objects, camera choreography, parametric based, and model driven systems is presented. In appendix A a short discussion of the various hardware components required for an animation recording environment is given.

Introduction

What is computer animation? The most general description of computer animation may be stated as follows: Computer animations may be created by using a rendering device that will produce consecutive frames consisting of relative changes in visual effects. These changes in effects may be due to either motion dynamics, such as time-varying position of rigid or non-rigid bodies, or update dynamics. Update dynamics is associated with a change in shape, color, transparency, structure, or texture. Other changes in effects can be accomplished by modifying the lighting characteristics or changing the camera position.

Animation Environments

1) Homogeneous Coordinate System

Before one can discuss the various animation techniques, it is essential to understand the basic principles of the environments in which these animations are performed. If one analyzes the transformations of a vertex point P(x,y,z) to P'(x',y',z') where P is represented as a row vector and

$P' = P + T$ (for the translations)
$P' = P * S$ (for scaling)
$P' = P * R$ (for rotations),

then one can see that translations are a linear transformation, and the scaling as well as the rotations are associated with scalar transformations. By using the homogeneous coordinate system, all three transformations can be treated as multiplications. This is accomplished by introducing the w coordinate where

$P(x,y,z) \to P(X,Y,Z,w)$ and $w \neq 0$.

The 3D Cartesian coordinates can then be represented as

$$x = X/w, \quad y = Y/w, \quad z = Z/w.$$

In computer graphics w is set equal to 1 to simplify computations. Hence the vertex point P in the homogeneous coordinate system can be represented as P(x,y,z,1).

2) Transformation Matrices

To transform an object in the homogeneous coordinate system, vertex points of an object are multiplied by a 4x4 matrix. The representative matrices for translations (T), scaling (S), and rotations (R) are shown below and will be referred to as such throughout this paper.

Translations

$$T(T_x,T_y,T_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Scaling

$$S(S_x,S_y,S_z) = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotations

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

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

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

Note: The scaling and rotation matrices are derived to transform vectors about the origin of a Cartesian coordinate system.

3) World and Local Coordinate Systems for Hierarchically Defined Scenes

To discuss these two coordinate systems, an example of a robot will be used. The robot consists of a base, two arms, and a claw. A graphical representation of the robot is shown in Figure 1 below. Figure 2 shows the hierarchical relationship of the various objects. One can see in Figure 2 that the top node is referred to as the "picture node". The "picture node" represents the entire scene to be displayed. The world coordinate system (WCS) defines the space of the entire scene, and many objects can be placed within it. The initial placement as well as any transformations of these objects will be performed with respect to the origin of the WCS. An artificial pivot point may be introduced into the scene to provide extended motion capabilities. This will be discussed in more detail below. The data structure associated with the picture node is called "FOREST_DEF". The data structure associated with the root node and the children nodes is referred to as the "NODE". Both of these structures as well as several other functions referred to later in this paper are available in the C-code reference listed at the end of this paper.

Referring to Figure 1, only one hierarchical object, the robot, which is defined in the scene, is shown. If additional objects were added, they would be attached to the picture node. The node which represents the entire object is called the "root node". Transformations of the "root node" may be performed either in the WCS or the local coordinate system (LCS). It is essential to allow the root node to be transformed in the WCS so that it can be moved throughout world space. In addition it is necessary to transform the object with respect to its own LCS moving about its own axes. Objects are modeled with respect to their own LCS, i.e., all points of the object are defined with respect to this coordinate system. In general the origin of the LCS coincides with the pivot point of the object. The pivot point may either be the center of rotation, center of mass, or any pre-defined point on the object.

In Figure 2, one can see that all other nodes, or objects, defining the characteristics of the robot, are linked in a hierarchical manner. These nodes are called children nodes. The node above a child node is called the parent node. Children nodes may have their own children attached to them. Children nodes may be transformed only in their own LCS. Without this restriction, the hierarchical linkage of the objects could be easily separated. Even with this restriction, problems may arise if care is not taken. It is generally necessary to introduce motion constraints on the individual nodes. For instance, an object that may be moved around a joint should not be allowed to translate; otherwise it will be separated from its parent node. The beauty of defining objects in a hierarchical manner is that children nodes may inherit the transformations as well as attributes from their parent.

Whenever transformations are performed in either the world coordinate or local coordinate system, respectively, world and local transformations are required. These will be discussed next.

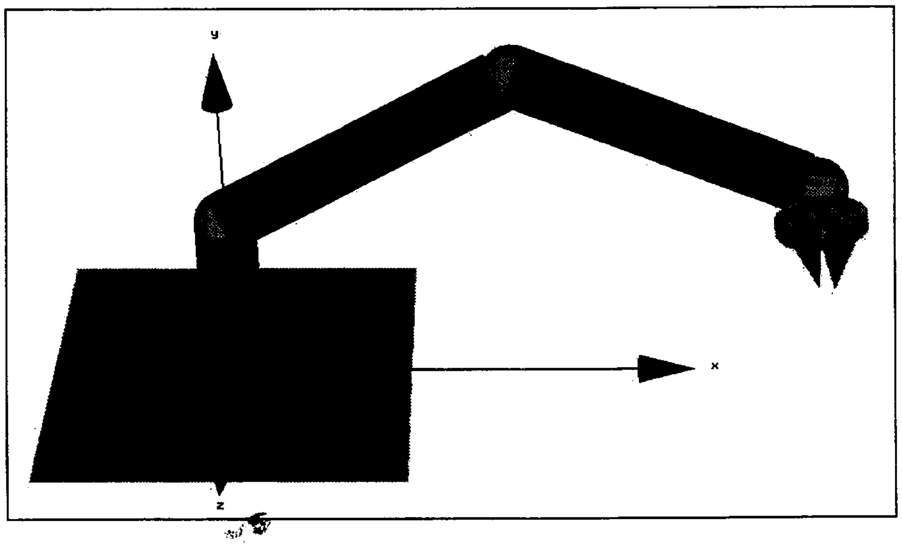

Figure 1: Display of a Hierarchically Linked Robot

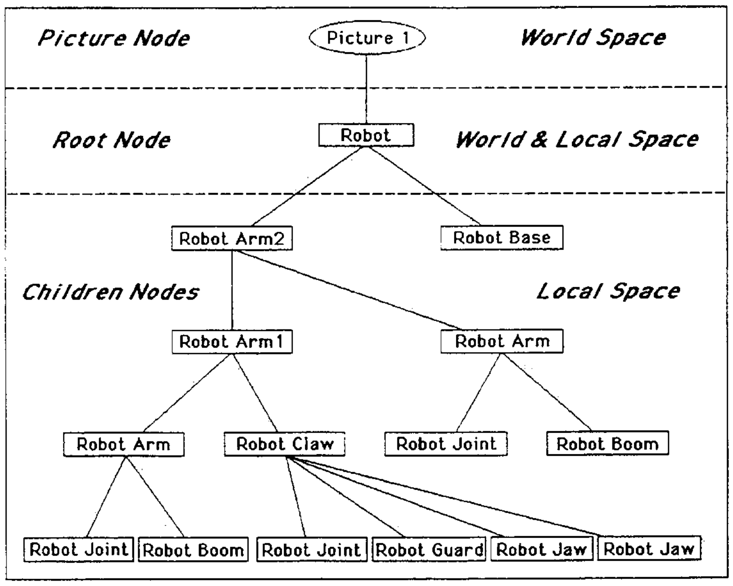

Figure 2: Hierarchical Diagram of the Robot

4) Transformations in World and Local Space

Whenever transformations are performed in either the LCS or the WCS, a set of rules are required to perform the matrix multiplications. The following techniques were implemented in an animation package developed at NASA Ames Research Center called Ames Research Center Animation Development Environment (ARCADE). This paper does not claim that the following method is the only way to perform transformations on hierarchically linked objects. This method, however, has produced the best results for the animation package.

All transformations in either world or local space require incremental update of the node transformations. An incremental change in motion of a selected node is added to the accumulated motion of this node.

4.1) Local Coordinate System

To describe transformations with respect to the LCS the following components need to be defined:

R_Δ = Incremental rotational matrix
S_Δ = Incremental scaling matrix
T_Δ = Incremental translational matrix
T_{LP} = Translations with respect to the local pivot point
M_L = Previously computed resultant transformation matrix in local coordinates
M'_L = New resultant transformation matrix in local coordinates

These components are used in the following matrix calculations:

$$M'_L = R_\Delta(\phi) * S_\Delta(s_x, s_y, s_z) * M_L$$
$$M'_L = T_\Delta(x_2, y_2, z_2) * T_{LP}(-x, -y, -z) * M'_L * T_{LP}(x, y, z)$$

For example, these matrix multiplications can transform an object (see Figure 3) from position P_1 to position P_2 via a set of translations, rotations, and scalings in the local coordinate system. To do this, a translation matrix must be established to translate the object from (A) to (B). Note that the scaling and rotation matrices defined previously are only valid for transformations about axis' origin. To transform the object with respect to its pivot point, the pivot point is translated to the origin (C), the rotations and scales can then be applied (D&E), and the pivot point is then translated back to its previous location. Now the translational matrix is applied to position the object to the point P_2. Appendix B provides sample C code performing these operations (see the function called "local_transform()").

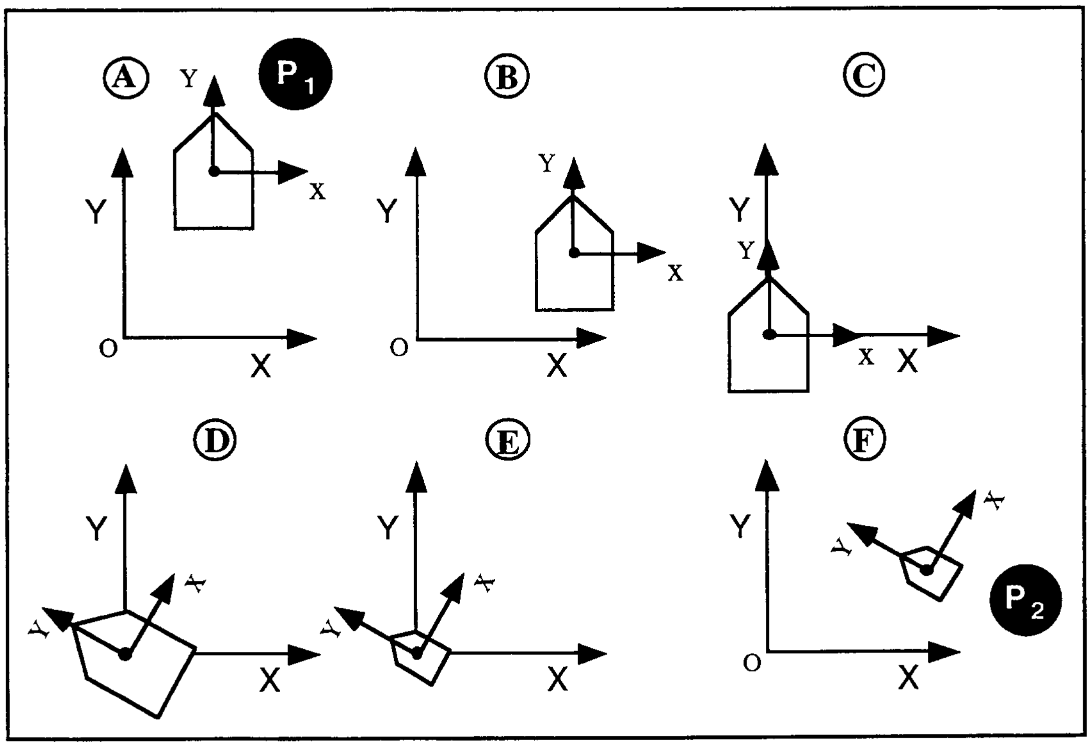

Figure 3: 2D Transformation with Respect to the Local Coordinate System

4.2) World Coordinate System

To describe transformations with respect to the WCS the following components need to be defined:

R_Δ = Incremental rotational matrix
S_Δ = Incremental scaling matrix
T_Δ = Incremental translational matrix
T_{WP} = Translations associated with the world pivot point
M_W = Previously computed resultant transformation matrix in world coordinates
M'_W = New resultant transformation matrix in world coordinates

$$M'_W = M_W * T_\Delta(x_2, y_2, z_2) * T_{WP}(-x, -y, -z) * R_\Delta(\emptyset) * S_\Delta(s_x, s_y, s_z) * T_{WP}(x, y, z)$$

For example, these matrix multiplications transform an object from position P_1 to position P_2 via a set of translations, rotations, and scales in the world coordinate system. Again a translation matrix may be established to translate the object from (A) to (B). Next the rotations and scales of the object may be computed about an arbitrarily defined pivot point in world space. Generally, this pivot point is set to equal the world axes' origin. However, to allow more motion flexibility it may be located anywhere in world space. The rotations and scales are then performed about this arbitrary pivot point. As mentioned earlier, all rotations and scales must be computed about the axes' origin. To assure this, the arbitrary pivot point is translated to the world axis origin (C), the rotations and scales can then be applied (D&E), and the arbitrary pivot point is then translated back to its previous location. At this point the translational matrix is applied to position the object to the point P_2. The C-code performing these operations can be found in the function called "world_transform()". Note in Figure 4, section E, that, whenever a scaling operation is performed on an object about an arbitrary world pivot point, a translation toward this pivot point is introduced. This can be very useful in animating imploding (bursting inward) objects.

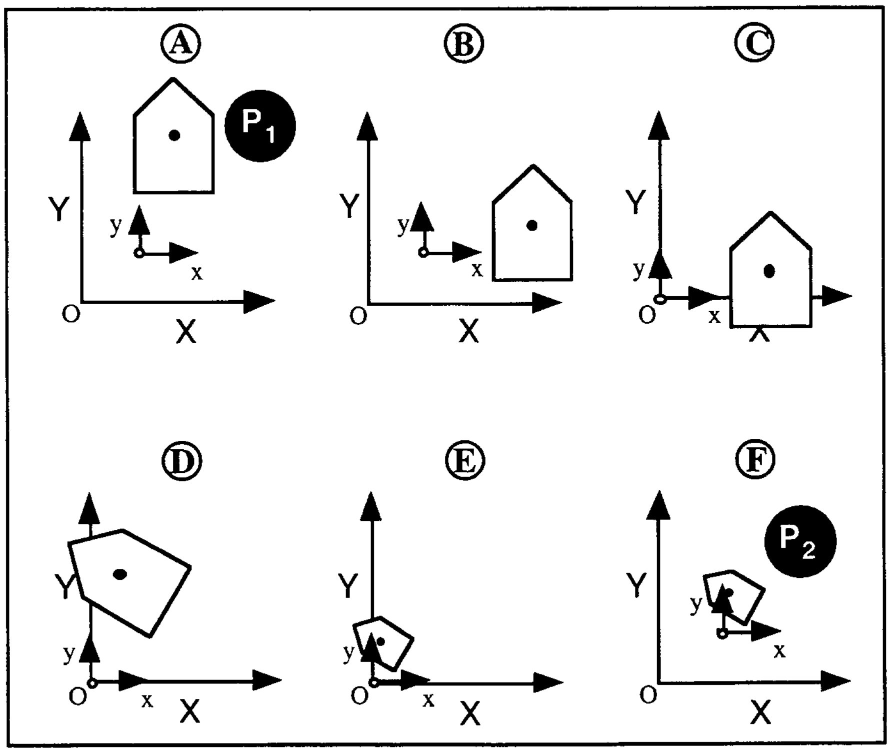

Figure 4: Transformation with Respect to the World Coordinate System

4.3) Switching Between World and Local Transformations

As mentioned above the root node of any object may be transformed with respect to either the WCS or the LCS. To assure continuity between switching from one space to the other, the object matrix must be converted to the respective coordinate space. Without this conversion, the object would not maintain its position (i.e., it would jump when switching between WCS -> LCS ->WCS). Simple matrix multiplications on the node matrix M must be performed to maintain continuity. The following general equation is used to switch from the WCS to the LCS or from the LCS to the WCS depending on the value of "direction."

$$M' = \text{direction} * (-T_{LP}) * M * \text{direction} * (+T_{LP})$$

where LCS -> WCS: direction = 1
WCS -> LCS: direction = -1

5) Key Frame Systems

Now that the fundamentals (e.g., coordinate systems, hierarchy, etc.) have been covered, it is time to move on to animation techniques. The most commonly used animation technique is called "key frame" animation. Key frame animation was originally developed by Walt Disney to create cartoons (reference #1). In these systems, skilled animators designed or choreographed images at a specific instance in time representing the animation. These images, the so called key frames, would then be interpolated by less skilled animators to complete a sequence of "in-between" frames. Animating the key frames along with the in-between frames resulted in the desired cartoon.

Within computer graphics, the key frames represent a 4x4 transformation matrix associated with an object (node). The in-between frames are then computed by the computer using a variety of interpolation techniques. The most frequently used interpolation techniques are linear, Hermite, splines, and quaternions, all of which are described in this section.

5.1) Key Frames in Computer Graphics

To create the desired key frames it helps to have an intuitive user interface to control the manipulation of scene objects. Selecting desired nodes from some sort of tree representation (see Figure 5) of the hierarchical object works quite well. Once a node has been selected, transformations can be used to establish a series of key frame matrices. For example, the manipulation of a hierarchically based robot object (see Figure 5) to create key frames (see Figure 6) could adhere to the following scenario. First the user would select the robot arm1 object (node at (A)) and perform a rotation about the z-axis. This matrix is then saved as the first key frame. Note that a translational motion constraint, to prohibit translations in either x, y, or z direction, should be imposed on any of these "arm" joint nodes. Translating these nodes would separate them from their parent node. It can be clearly seen in Figure 6 that the children nodes inherit the transformation of the applied matrix. Next a transformation matrix of the robot claw object (node at (B)) and the transformation matrix of the robot arm2 object (node at (C)) are saved, each creating their own key frame matrix. By interpolating the motion of the key frame matrices and displaying the object with each interpolated matrix, a nice motion of the robot can be visualized.

The C-code to display the current picture as well as display the nodes of a tree is shown in the functions "display_scene()" and "display_tree()". The function "display_tree()" takes care of the inheritance issues associated with hierarchical objects. This is accomplished by using the function calls pushmatrix, popmatrix and pushattributes and popattributes which preserve or delete the matrix and attribute information on a stack accordingly.

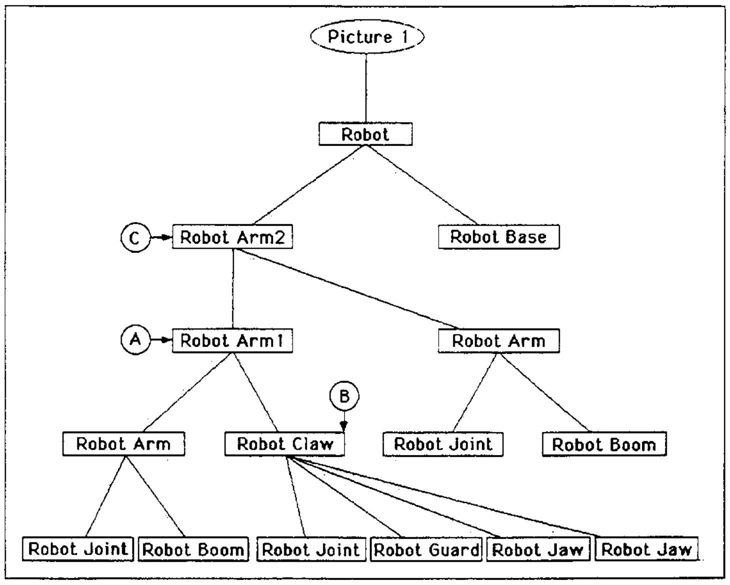

Figure 5: Selecting Nodes for an Animation

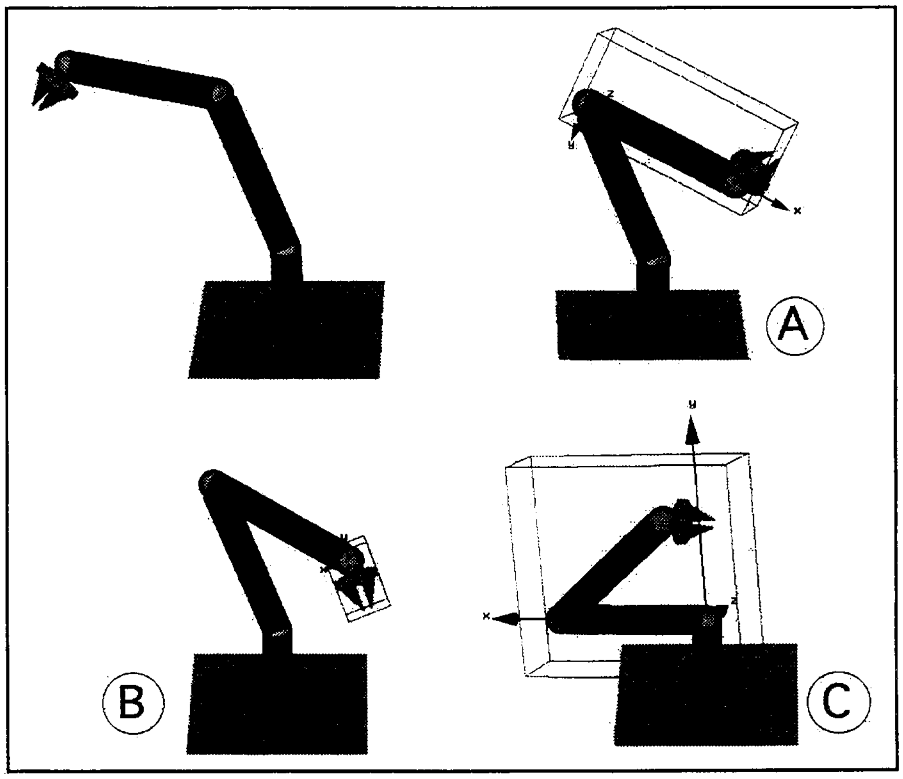

Figure 6: Selected Key Frames for an Animation

5.2) Controlling the Kinematics of the Animation

The placement of the key frames with respect to time determines the kinematics (motion) of an object. Key frames, which are placed closer together with respect to time, produce a faster motion than key frames that are placed farther apart. It is the opinion of the authors that an interactive user interface should be used to easily facilitate control of the kinematics in an animation. As an example the user interface developed for the animation package ARCADE is shown in Figure 7.

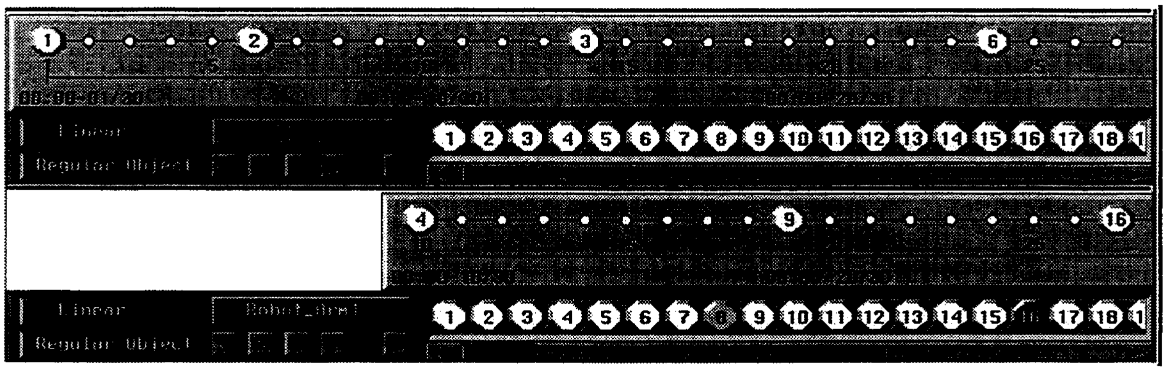

Figure 7: Animation Strand within ARCADE

The user interface shown in Figure 7 displays a graphical representation of the time line for each node that will be animated. Each displayed time line is called a "strand." Besides having control features, the strand also contains a key frame "bucket" where graphical representations, i.e., icons, of the key frames are displayed. A user can place these matrix icons onto the strand of the respective node to control the kinematics of the animation.

5.3) Interpolating the Key Frames

Key frame interpolation is an essential part of animation because it allows the animator to define a small number of strategic frames that the software then uses to create numerous in-between frames, thereby producing a smooth animation. There are several interpolation techniques possible (reference #1, #2, #3), however only a few of them are suited to produce an appropriate animation. Figure 8 shows possible candidates for an interpolation method. Each curve in this figure will be discussed in more detail.

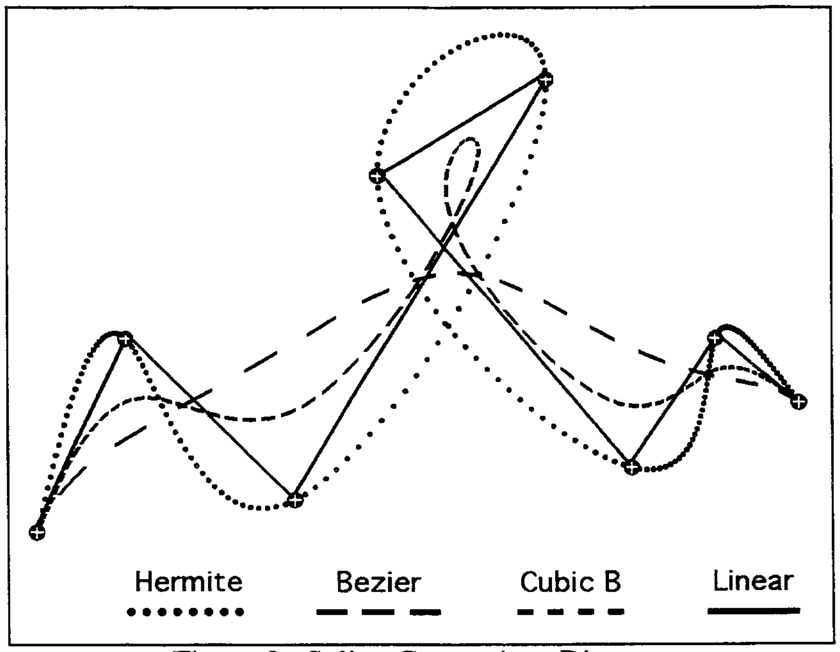

Figure 8: Spline Comparison Diagram

5.3.1) Linear Interpolation

The easiest interpolation technique is linear interpolation. Linear interpolation generates continuous motion, however it also generates first- and second-order derivative discontinuities at the key frames. As a result the animation experiences abrupt changes in direction at the key frames which give the animation a mechanical look. Sometimes this is the desired effect especially with very rigid bodies; however, a much smoother motion is usually desired. One advantage of the linear interpolation is that it guarantees that the animated path will go through the set of selected key frames.

5.3.2) Hermite Interpolation

A much more desirable interpolation technique is the Hermite interpolation. The Hermite interpolation guarantees derivative continuities, hence producing the desired smooth motion. As with linear interpolation, the animated path of the Hermite interpolation passes through the set key frames. One potential disadvantage of the Hermite interpolation is the exaggerated path it produces, which can bee seen in Figure 8. The top portion of the loop swings out beyond the region of the key frames. This region may, however, no longer be in the viewing volume (visible on the screen), and the object could temporarily disappear from the displayed scene.

5.3.3) Cubic B-Spline

Splines in general prevent objects from disappearing from the viewing volume because splines are confined by the convex hull of the set of control points. As long as each control point (key frame) is in the viewing volume, at least a portion of the object will always be visible in the scene. The cubic B-spline produces derivative continuities at the control point and enjoys the properties of being confined within the convex hull. The animation path of the cubic B-spline generally does not pass through the control points, which may create a problem. This can be overcome by doubling or tripling the particular vertex point as was done at the beginning and end control point in Figure 8. This action considerably reduces the motion's smoothness if performed on other than the begining and end control points and it increases the number of required computations.

5.3.4) Bezier Curve

From the curves shown in Figure 8, the Bezier curve is the least desirable since it completely dampens the animated path. The loop specified by the key frames shown in Figure 8 is "washed out" when using the Bezier curve to compute the in-betweens; consequently, it is rarely used for key frame animations.

5.3.5) Beta Spline

The Beta spline (see Figures 9 and 10) has many of the same advantages as the cubic B-spline, such as continuous derivatives at the control points and the spline being confined within the convex hull. Furthermore, the Beta spline has unique controlling parameters which provide total control of the curve's shape, making this spline a nice candidate for computing the in-betweens. The parameters that control the shape of this curve are the bias (β_1) and the tension (β_2). In Figure 9 the tension is set to zero and the bias is changed. One can clearly see that when $\beta_1 = 0$, a linear interpolation is obtained. When $\beta_1 = 1$, the Beta spline is equal to the cubic B-spline. As the bias is increased the actual shape of the curve can be controlled by shifting toward the left and approaching the control points. In Figure 10 the bias was set to one and the tension was modified. As the tension is increased the curve is stretched vertically to approach the control points.

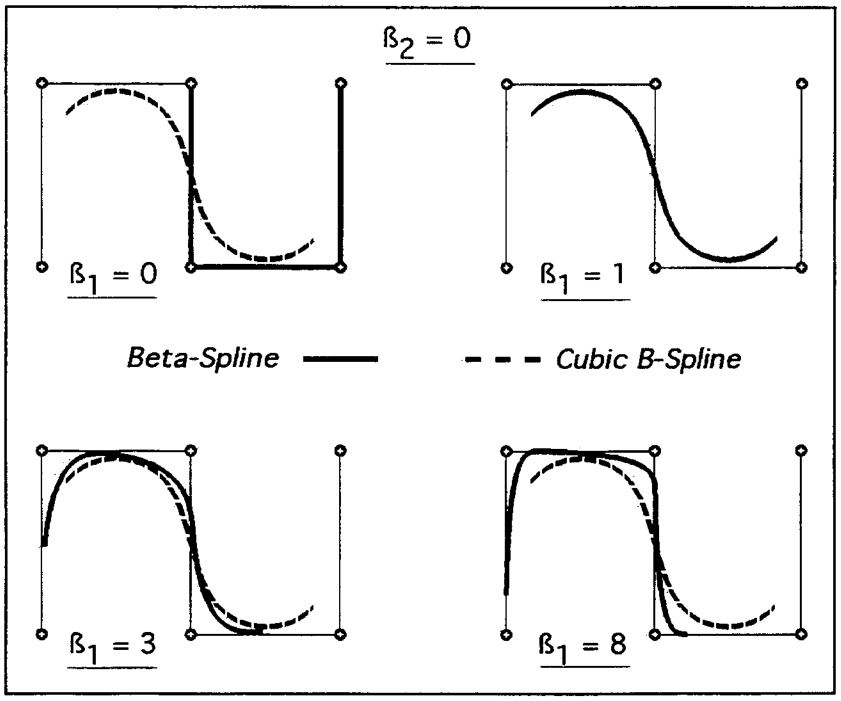

Figure 9: Evaluation of the Beta Spline (modifying the bias β_1)

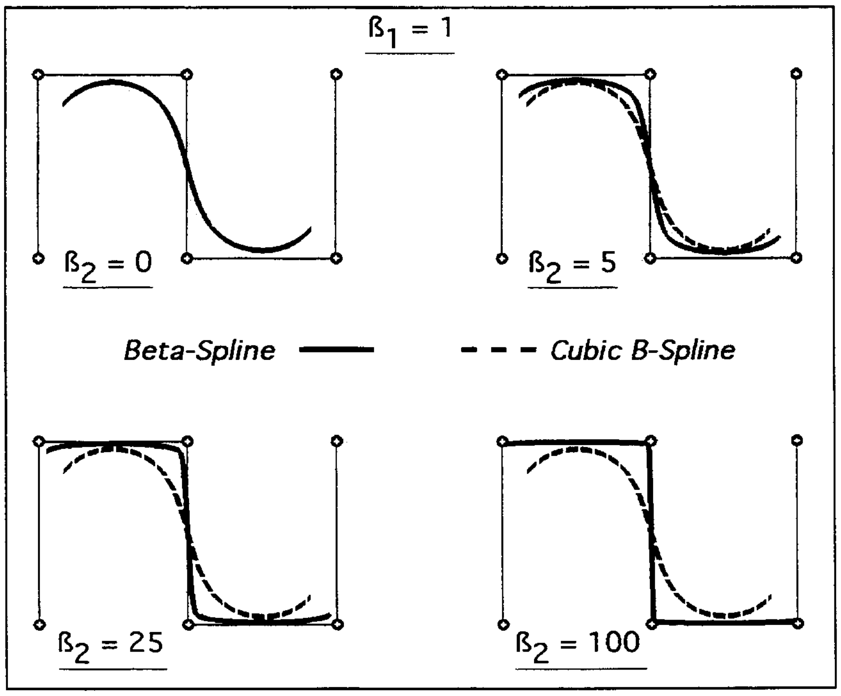

Figure 10: Evaluation of the Beta Spline (modifying the tension β_2)

5.3.6) Quaternions

The above mentioned techniques work very well for computing the in-betweens for the translation and scaling parameters, but not for the rotation parameters. In the past the Euler angles, the angles defining the 4x4 rotational matrix R above, have been used extensively for animating orientation; however, interpolating Euler angles leads to unnatural interpolation of the angles. For this reason quaternions should be used as the interpolation technique for rotations (reference #1 , #2). In previous discussions the 4x4 matrix M was computed for either the WCS or the LCS. This matrix was computed by a series of matrix multiplications of translations, rotations, and scales. In order to perform the quaternion interpolation it is necessary to extract the rotational components from the matrix M. Since the scaling components S_x, S_y, and S_z are multiplied by the rotational components, this may not be a trivial task. If $S_x = S_y = S_z$ then the task of extraction is simplified. However if they are not equal, shearing matrices must be introduced to extract the scaling components. Reference 4 provides a detailed explanation of this procedure. In general the matrix M must be separated into five different matrices, three of them are shearing matrices (see below). The in-betweens associated with shearing matrices, the scaling matrix as well as the translation parameters may then be computed either using splines, or the hermite or linear interpolation. The normalized 3x3 rotational matrix can now be converted to its respective quaternion representation. The quaternion in-betweens for the set of quaternion key frames may be computed using conventional interpolation techniques. This produces a spherical linear interpolation of the key frames. The in-betweens as well as key frames need to be converted back to their respective rotational matrix representations. At this point the related in-betweens for the translations, rotations, scales, and shears must be multiplied to produce the respective in-between matrix M used in the animation.

$$M = \overset{\underline{\text{Scaling}}}{\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \overset{\underline{\text{Shear}_{xy}}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ S_{xy} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \overset{\underline{\text{Shear}_{xz}}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ S_{xz} & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \overset{\underline{\text{Shear}_{yz}}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & S_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \overset{\underline{\text{Rotation \& Translation}}}{\begin{bmatrix} \begin{bmatrix} R[3X3] \end{bmatrix} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ T_x \; T_y \; T_z & 1 \end{bmatrix}}$$

C-code for the above mentioned routines is available via several functions (**normalize**(), **QuatFromMatrix**(), **QuatToMatrix**(), **Interpolate**(), **slerp**()) in Appendix B.

6) Cycled Object Animation Technique

Another animation technique used in conjunction with the key frame systems is the cycled object animation technique (e.g., display of consecutive frames of time dependent objects or data), which is used extensively in scientific visualization. A large set of objects, representing either time dependent data or a display of varying physical object characteristics, such as position, shape or color, may be presented by the scientist by this method. Displaying each object of the entire set over time may produce an animation containing vast amounts of data. Generally key frames are established to define the position of the cycled objects within world space. Initially a skip range is established which determines the objects from the set that will be displayed. Next the set of objects are cycled with respect to time using several simple techniques. A bounce cycle may be used where the cycle begins the display with a pre-defined start object and incrementally displays the next object based on the skip range. Once the last object is reached, the cycle begins with the last object and cycles forward to the first object where the cycle starts again.

Bounce cycle: $obj_s \rightarrow obj_i \rightarrow obj_n$; $obj_n \rightarrow obj_i \rightarrow obj_s$; ....

The looping cycle is similar to the bounce cycle, however rather than continuing with the last object when the last object has been displayed, the cycle continues with the first object.

Looping cycle: obj_s -> obj_i -> obj_n; obj_s -> obj_i -> obj_n; ....

The growth cycle allows the objects to be accumulatively displayed.

Growth cycle: obj_s -> $obj_s + obj_{s+1}$ -> $obj_s + obj_{s+1} + obj_{s+2}$ -> ....

7) Camera Choreography

Besides manipulating data, animations can be created by keeping the data static and manipulating the camera (reference #2). This type of animation allows an animator to "fly" about the data and view it from various locations in world space. Zooming and panning techniques may be used to produce other views. Several different parameters may be modified for camera choreography (refer to Figure 11). The camera viewpoint or twist, which establishes the actual physical location and orientation of the camera in world space, may be modified. In addition, the reference point of the camera, the location where the camera is looking at, may be changed. When creating animations with the camera, start and end any motion slowly (slow-in & slow-out). In addition, jerky motions throughout the animation should be avoided.

An advantage of camera choreography is that the data does not have to be manipulated, it only needs to be viewed. For this reason the time required to produce an animation may be reduced considerably.

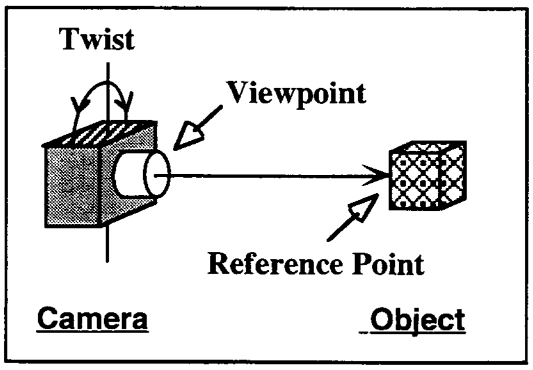

Figure 11: Camera Choreography

8) Other Animation Techniques

Besides using key frames and camera choreography, there are many other techniques that are available to enhance the visualization experience. In general these other techniques are used in conjunction with the key frame and camera choreography animations. They include parametric systems, P-curves, animating object attributes such as color and transparency, as well as introducing special objects used to visualize a range of phenomena. These special objects may be contour lines and sweeps; particles, streamlines and vectors; ribbons and streamers; as well as isosurfaces and cutting planes.

8.1) Parametric Systems

Parametric systems are an extension of the key framing technique. In some applications, while it is tedious to generate key frames, the 4x4 transformation matrix can be easily created by procedures (functions). In this case, an object would be linked to one or several motion functions, where the motion is determined by a set of time dependent parameters. These parameters may be modified through an interactive user interface. A simple example of a parametrically based system would be the rotation of propellers of an aircraft. By specifying the revolutions per minute as a function of time, the rotational matrices required for the animation can automatically be calculated by the computer. It would be a very tedious job to create such an animation via key frames alone. Another significant aspect of this system, which cannot be achieved by key frames, is to incorporate some constraint into the system. For example, two colliding objects should not penetrate each other.

8.2) P-Curve

The P-curve is a parametric representation of the motion or any attribute associated with an object. Rather than using functions as mentioned in the parametric systems above, the P-curve establishes the animation via a graphical user interface. The positional coordinates of the motion x(t), y(t), and z(t) for instance are described by spline curves (see the 2-D analysis, Figure 12). By specifying the position (or attribute) on the curve at each frame, the animation is created. Obviously, the graphical user interface plays an important role in controlling animations. By editing the parametric curve, animations with different paths or attributes can be ceated. (For details see references #2, #5)

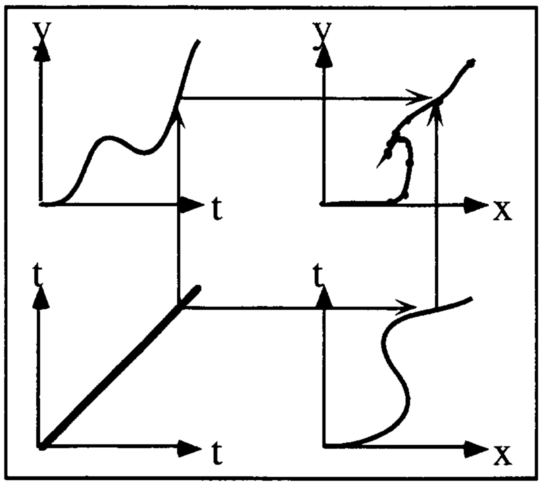

Figure 12: P-Curve

8.3) Animation Object Attributes (Color and Transparency)

Color is used extensively in scientific visualization to analyze the dynamic range and distinguish particular values of a data set. Colors are mapped to a specific value or a range of values which allow researchers to have a clearer understanding of the topology of their data. Transparency is used extensively to provide a view of data hidden behind data of another plane.

8.4) Animating Special Objects

8.4.1) Contour Lines and Sweeps

Contour lines are used to visualize the threshold of a range of data sets in a particular plane. Data values equal to user defined threshold settings are graphically linked via lines which may or may not be colored by the magnitude of the data. The contour lines may be shown either by themselves or with the remaining data of the plane that has already been colored. Contour sweeps can be used to demonstrate the progressive change in data by visualizing the contour lines of various data planes. If time varying data is available, then these contour lines are set into motion depending on the dynamics of the data.

8.4.2) Particles, Streamlines, and Vectors

Particles and streamlines are used extensively in the visualization of computational fluid dynamics. To understand the path and the dynamics of the flow, particles are "injected" in the computed flow. Streamlines are obtained by connecting the paths along which the particles travel. Vectors may also be used to demonstrate the magnitude and direction of the flow. Vortices and other physical flow phenomena can clearly be shown using these techniques. Particles, streamlines, and vectors may be colored by the magnitude of the particular physical parameter to be displayed.

8.4.3) Ribbons and Streamers

Visualization using ribbons and streamers is very similar to visualizing particles and streamlines. The primary difference is ribbons and streamers produce a segmented surface representation of the flow.

8.4.4) Isosurfaces and Cutting Planes

These two techniques are used extensively in scientific visualization. Isosurfaces represent the three dimensional surface asociated with a particular value in the data set. As an example, isosurfaces are used to show the volumetric surface of a particular pressure or temperature. Data hidden by an isosurface may be displayed by either making the isosurface transparent or by using cutting planes. Cutting planes allow a scientist to slice discrete planes within the isosurface exposing any hidden visualized data.

Advanced Animation Techniques

9) Soft Object Animation

Shape distortions in animations are generally used to highlight dynamic actions of a scene, such as objects experiencing an impact (reference #6, #7, #8). Generally the shape distortions are completely arbitrary and under the control of the animator. The exception to this is when shape distortions are created from physically based models, i.e., forces acting on an object which produce the distortions. Simply speaking, shape distortions can be achieved by either moving the vertices of the polygonal model or by modifying the control points of a representative parametric surface of the model. When changing vertices, the polygonal resolution of the model may constrain the nature of the deformation. Models with low polygonal resolution may experience shape degradation. A better method for generating shape deformation is to modify the control points of the parametric surface.
By changing the coefficients of the basis functions of the parametric surface, the functional description of the surface is altered. Bezier or B-spline patches are well suited for parametric surface representation. This method can produce deformations of any complexity while maintaining a smooth surface of the model.

10) Animation of Particle Systems

Many complex models may be animated by animating representative particles of the particular model (reference #9). Animating rain drops, the flow of water, a flock of birds or even a table cloth are among the typical applications. There are two different methods for animating particle systems, namely, the scripted and non-scripted systems. The latter provides for much more realistic physically based animations.

10.1) Scripted Particle Systems

In scripted systems the objects, which are represented by particles, may be animated by user defined scripts controlling the particles' dynamics and appearance. A frame by frame generation of the animation would be accomplished by the following method:
First, new particles are generated and injected into the system. The number of particles released, N(t), may be computed by

$$N(t) = M(t) + \text{rand}(r)V(t)$$

where M(t) is a mean number of particles as a function of time. The product rand(r)V(t) would represent a random variable of variance, as a function of time. Each new particle would be assigned its own initial conditions such as position, velocity, direction, size, shape, color, transparency, and its lifetime. Any particle which exceeds its lifetime will be extinguished from the scene. The particles would move according to their user defined scripts based on dynamic constraints. At this point the particles can be rendered.

10.2) Non-scripted Particle Systems

During the last three years increased computational power as well as sophisticated computer graphics hardware has made it possible to produce very advanced animations using physically based modeling of particle systems. The motion of the particles is completely based on Newtonian mechanics. Various types of forces may act on these particles, which may either be unary forces (gravity, drag, etc.) that act independently on each particle, n-array forces (springs) that are applied to a fixed set of particles, or forces of spatial interaction (attraction, repulsion) that may act on all or pairs of particles. Mass-spring systems may be created that approximate the motion of rigid or elastic bodies as well as fluids. In this system the mass of a body may be lumped into a collection of mass points. The motion between these mass points may be approximated by a set of springs. By changing the spring constant one can modify the viscosity of a fluid or the elasticity of an object. Several papers have been written (reference #4) over the last three years on how to incorporate these techniques into computer graphics animation.

10.3) Behavioral Animation

Behavioral animations may be used to simulate biological systems (reference #10). Once again the concept is based on particle systems. With this method particles are no longer independent, they interact with each other to simulate various flocking mechanisms. The flocking of the entire system is controlled by global positions or global direction vectors. The flocking individuals are controlled by the following criteria. A flock mate will always avoid a collision with a nearby flock mate, while maintaining the closest proximity to the other flock mate. In addition a flock mate will try to maintain the velocity of the nearby flock mates. Very nice animations of a flying flock of birds have been created using this method.

Summary

11) Although its most common application is in the comercial entertainment industry, computer animation has become an important tool for visualizing complex and time dependent data in recent years. This paper details the animation fundamentals and provides an overview of most commonly used animation techniques in Scientific Visualization. Some techniques such as the key frame systems were discussed in more detail because of their underlying importance associated with animations. Due to increased computational power and sophisticated rendering devices, advances in animation techniques, such as physically based modeling, animation will play a more important role in simulation and visualization of the real physical world.

Acknowledgments

The authors owe the warmest thanks to Eric Hibbard, who read the text with meticulous care. His extensive and penetrating comments have been most important in improving the paper in all its aspects. The authors are very grateful to Randy Kaemmerer for commenting on the draft.

References

1) Allen Watt, Mark Watt, "Advanced Animation and Rendering Techniques," *ACM Press/Addison-Wesley*, New York, '92;
2) J. Foley, A. van Dam, S. Feiner, J. Hughes, "Computer Graphics, Principles and Practice," Second Edition, *Addison-Wesley*, Nov, '91;
3) R. Bartels, J. Beatty and B. Barsky, "An Introduction To Splines For Use In Computer Graphics & Geometric Modeling," *Morgan Kaumann*, Los Altos, CA., '87;
4) J. Arvo, "Graphic Gems II," Academic Press, San Diego, CA., '91;
5) R.M. Baecker, "Picture Driven Animation," SJCC, *AFIPS Press*, 1969;
6) M. Gascuel, "An Implicit Formulation for Precise Contact Modeling," *Computer Graphics*, pp. 313-20, August 1993. Proceeding of SIGGRAPH '93, held in Anaheim, CA, 1-6 August, 1993;
7) D. Baraff and A. Witkin, "Dynamic Simulation of Non-penetrating Flexible Bodies" *Computer Graphics*, pp. 303-308, July 1992, Proceeding of SIGGRAPH '92, held in Chicago, IL, 26-31 July, '92;
8) D. Metaxes and D. Terzopoulos, "Dynamic Deformation of Solid Primatives with Constraints" *Computer Graphics*, pp. 309-312, July 1992, Proceeding of SIGGRAPH '92, held in Chicago, IL, 26-31 July, '92;
9) M. Kass, A. Witkin, D. Baraff and A. Barr, "An Introduction to Physically Based Modeling," *1993 Siggraph Course Notes 60,* Anaheim, CA;
10) C. W. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioural Model," *Computer Graphics*, pp. 25-34, Proceeding of SIGGRAPH '87;

Appendix A
(Typical Hardware Components)

In general, to produce an animation, naturally some type of display device is required to produce the image on a monitor. In addition a device is required to convert the RGB signal(s) from the display device into a recordable signal like RS-170A. A frame controller is required to accept commands from the computer to cause one or more frames of the animation to be recorded. Finally, a recorder is used to record the image data on some media.

The basic components used at NASA Ames Research Center are listed as follows: The display device used is either a workstation or an X-terminal. A scan converter, such as a Lyon Lamb real time scan converter (RTC), converts the RGB signal to a recordable signal. A Lyon Lamb miniVAS controls the sequence of frames. A Betacam SP deck is used as the mastering deck, however, any recording device that support frame accurate recording may be used (e.g., U-matic SP, VHS, S-VHS, etc.).

Appendix B
(C Code)

```
/**************** Hierarchical Structures ***************/
/***
```

```
  Tree structure
***/
struct node {
          /*
            .... node data
          */
          PIVOT_PTR          node_pivot;  /* Local pivot point of the node */
          MATRIX_PTR         node_matrix; /* Actual drawing matrix        */
          MATRIX_PTR         original_mat; /* Will never be changed       */
          MATRIX_LIST        animate_mat; /* List of saved Keyframes      */
          struct node     *next;
          struct node     *parent;
          struct h_node    {
                           int          length;
                           struct node *first, *last;
                           } *leaves;
        };

typedef struct node       NODE;
typedef struct h_node   H_NODE;
typedef H_NODE         *LEAVES;
typedef NODE           *TREE;

/***
  Linked list of picture structures (forest of trees)
***/
struct forest_def
     {
     /*
         ... picture data
     */
     TREE              tree_address; /* Pointer to a tree structure  */
     Gen_matrix        pict_mat;     /* Camera transformation matrix */
     CAMERA_LIST      camera;       /* List of camera keyframes     */
     struct forest_def  *previous;
     struct forest_def  *next;
     };

typedef struct forest_def      FOREST_DEF;
typedef FOREST_DEF          *FOREST;

/***************** Initializing the Node Matrix ***************/

/***
  Create the Node Matrix.  If an ARCgraph matrix exists, use it as a node matrix.
***/
void init_node_matrix( TREE tree_node )
{
  if ( tree_node->original_mat != NULL )
    copy_matrix(*(tree_node->original_mat),*(tree_node->node_matrix));
  else
    copy_matrix( Identity, *(tree_node->node_matrix) );
}
```

```
/*********** Computing the World Transformation Matrices **********/

/***
  Compute the world transformations
***/
void world_transform()
{
  /*
    COMPUTE THE NODE MATRIX
  */
  pushmatrix();
  loadmatrix( Identity );
  /*
   translate the world pivot point to the world origin (0, 0, 0)
  */
  translate( active_node->node_pivot->wrld_piv_pnt[0],
          active_node->node_pivot->wrld_piv_pnt[1],
          active_node->node_pivot->wrld_piv_pnt[2] );
  /*
   incremental rotation about the desired axis of the world origin
  */
  if ( xrot_flag )
    rot( xrotval, 'x' );
  else if ( yrot_flag )
    rot( yrotval, 'y' );
  else if ( zrot_flag )
    rot( zrotval, 'z' );
  /*
   scale about the desired axis of the world origin
  */
  if ( xscale_flag )
    scale( 1.0 + xscaleval, 1.0, 1.0 );
  else if ( yscale_flag )
    scale( 1.0, 1.0 + yscaleval, 1.0 );
  else if ( zscale_flag )
    scale( 1.0, 1.0, 1.0 + zscaleval );
  else if ( xyzscale_flag )
    scale( 1.0 + xscaleval, 1.0 + yscaleval, 1.0 + zscaleval );
   /*
   translate back to the local origin
  */
  translate( -active_node->node_pivot->wrld_piv_pnt[0],
             -active_node->node_pivot->wrld_piv_pnt[1],
             -active_node->node_pivot->wrld_piv_pnt[2] );
 /*
   perform a local transformation
  */
  if ( translate_flag )
    translate( xval, yval, zoomval );
  /*
   multiply the old accumulated matrix to get the new resultant matrix
  */
  multmatrix( *(active_node->node_matrix) );
  /*
```

```
    SET THE RESULTANT NODE MATRIX
  */
  getmatrix( *(active_node->node_matrix) );
  popmatrix();
}

/*********** Computing the Local Transformation Matrices **********/

/***
  Compute the local transformations
***/
void local_transform()
{
  /*
    COMPUTE THE NODE MATRIX
  */
  /*
   multiply the local node matrix
  */
  pushmatrix();
  loadmatrix( *(active_node->node_matrix) );
  /*
   incremental rotation about the axis of the local pivot point
  */
  if ( xrot_flag )
    rot( xrotval, 'x' );
  else if ( yrot_flag )
    rot( yrotval, 'y' );
  else if ( zrot_flag )
    rot( zrotval, 'z' );
  /*
   incremental scale about the axis of the local pivot point
  */
  if ( xscale_flag )
    scale( 1.0 + xscaleval, 1.0, 1.0 );
  else if ( yscale_flag )
    scale( 1.0, 1.0 + yscaleval, 1.0 );
  else if ( zscale_flag )
    scale( 1.0, 1.0, 1.0 + zscaleval );
  else if ( xyzscale_flag )
    scale( 1.0 + xscaleval, 1.0 + yscaleval, 1.0 + zscaleval );
  /*
   get the local node matrix
  */
  getmatrix( *(active_node->node_matrix) );
  popmatrix();
  /*
   incorporate the total linear translation
  */
  pushmatrix();
  loadmatrix( Identity );
  /*
   translate the local pivot point to the world origin (0, 0, 0)
  */
  translate( active_node->node_pivot->pivot_pnt[0],
             active_node->node_pivot->pivot_pnt[1],
             active_node->node_pivot->pivot_pnt[2] );
  /*
   multiply the local node matrix
  */
  multmatrix( *(active_node->node_matrix) );
  /*
   translate back to the local origin
  */
  translate( -active_node->node_pivot->pivot_pnt[0],
             -active_node->node_pivot->pivot_pnt[1],
             -active_node->node_pivot->pivot_pnt[2] );
  /*
   perform a local transformation
  */
  if ( translate_flag )
    translate( xval, yval, zoomval );
  /*
   get the resultant node matrix
  */
  getmatrix( *(active_node->node_matrix) );
  popmatrix();
}

/*********** Switching Between World and Local Space **********/

/***
  Toggle between transforming the objects about the world coordinate axes or about the
   objects' local coordinate axes.
***/
void switch_coord_system( int direction )
{
  /*
    From world to local: direction = -1,
    From local to world: direction =  1.
  */
  pushmatrix();
  loadmatrix( Identity );
  translate( direction * active_node->node_pivot->pivot_pnt[0],
             direction * active_node->node_pivot->pivot_pnt[1],
             direction * active_node->node_pivot->pivot_pnt[2] );
  multmatrix( *(active_node->node_matrix) );
  translate( -1.0 * direction * active_node->node_pivot->pivot_pnt[0],
             -1.0 * direction * active_node->node_pivot->pivot_pnt[1],
             -1.0 * direction * active_node->node_pivot->pivot_pnt[2] );
  getmatrix( *(active_node->node_matrix) );
  popmatrix();
}

/*********** Displaying the Current Picture Node **********/

/***
  Display the entire current picture (scene)
```

```
***/
void display_scene()
{
 /*
   Clean the matrix stack
 */
 pushmatrix();
 /*
   Make the viewing volume
 */
 make_viewing_volume();
 /*
   Set the lighting environments
 */
 set_lighting environment();
 /*
   Apply global translation and scaling
 */
 translate( cur_picture->pict_mat.xpos, cur_picture->pict_mat.ypos, -32.0 );
 scale( cur_picture->pict_mat.global_scale[0],
        cur_picture->pict_mat.global_scale[1],
        cur_picture->pict_mat.global_scale[2] );
 /*
   Display each node of the current picture with its appropriate transformations.
 */
 display_tree( cur_picture->tree_address );
 /*
   Restore the original matrix
 */
 popmatrix();
}
```

/***** ***Displaying all of the Nodes Associated with the Current Picture Node*** *****/

```
/***
  Display the tree nodes using a preorder tree traversal.
***/
void display_tree( TREE tree_node )
{
  TREE          temp;

  /*
    Push the node matrix and attributes
  */
  pushmatrix();
  pushattributes();
  /*
    Apply current node transformations if present
  */
  if ( tree_node->node_matrix )
    multmatrix( *(tree_node->node_matrix) );
  /*
    Draw the node object of the tree with its associated attributes
  */
```

```
   draw_the_node( tree_node );
   /*
     Pretraverse the tree
   */
   if ( tree_node->leaves != NULL )
     {
     temp = tree_node->leaves->first;
     while( temp != NULL )
         {
         display_tree ( temp );
         temp = temp->next;
         }
     }
   /*
     Restore the original node matrix and attributes
   */
   popmatrix();
   popattributes();
}
```

/********* ***Computing the Animation Inbetweens for the Saved Keyframes*** *******/

```
/ * Author:  Kenneth  Hu
 * Date:   11/1/93
 */
#include <stdio.h>
#define X  0
#define Y  1
#define Z  2
#define W  3
#define EPSILON 1.0e-8
#define HALFPI  1.570796326794895
typedef float Tmat[4][4];
typedef float Vec[3];
typedef float Quat[4];

/***
    Extract the scaling parameters from the general 4x4 transformation matrix.
    Input is the 4x4 matrix m which contains translational, rotational, and scaling
    parameters.
    Output is a 4x4 matrix only containing the rotations and translations.  In addition
    the scaling vector and the shear parameters are obtained.
***/
void normalize( Tmat m, Tmat mat, Vec scale, Vec shear)
{
    float sxy, sxz, syz;
    int i;

    scale[0] = fsqrt ( m[0][0]*m[0][0]+m[0][1]*m[0][1]+m[0][2]*m[0][2]);
    if ( scale[0] < EPSILON) {
       printf("WARNING: singular matrix, can not be processed\n");
       return;
    }
    for ( i=0; i<3; i++) mat[0][i] = m[0][i]/ scale[0];
```

```
sxy = 0.0;
/*
    Perform a dot product on the first row and the second row
*/
for ( i=0; i<3; i++)
   sxy += ( mat[0][i]* m[1][i]);
/*
   Subtract first row component from second row
*/
for ( i=0; i<3; i++)
   mat[1][i]=m[1][i] - sxy*mat[0][i];
scale[1] = fsqrt(mat[1][0]*mat[1][0]+mat[1][1]*mat[1][1]+
             mat[1][2]*mat[1][2]);
if ( scale[1] < EPSILON) {
   printf("WARNING: singular matrix, can not be processed\n");
   return;
}
for (i=0; i<3; i++) mat[1][i] /= scale[1];
shear[0] = sxy/scale[1];
sxz = 0.0;
syz = 0.0;
/*
    Perform a dot product on the first row and the third row
*/
for ( i=0; i<3; i++) sxz += ( mat[0][i] * m[2][i]);
/*
    Perform a dot product on the second row and the third row
*/
for ( i=0; i<3; i++) syz += ( mat[1][i] * m[2][i]);
/*
    Make the third row perpendicular to the first two rows
*/
for ( i=0; i<3; i++)
   mat[2][i] = m[2][i] - sxz * mat[0][i] - syz * mat[1][i];
scale[2] = fsqrt( mat[2][0]*mat[2][0] + mat[2][1]*mat[2][1] +
              mat[2][2]*mat[2][2]);
if ( scale[2] < EPSILON) {
   printf("WARNING: singular matrix, can not be processed\n");
   return;
}
/*
   Normalize the third row
*/
for ( i=0; i<3; i++) mat[2][i] /= scale[2];
/*
   Scale the shear factor
*/
shear[1] = sxz/scale[2];
shear[2] = syz/scale[2];

for ( i=0; i<4; i++){
   mat[i][3] = m[i][3];
   mat[3][i] = m[3][i];
}
```

```
}

/***
   Compute the quaternions from the normalized rotational matrix
***/
void QuatFromMatrix( Tmat mat,Quat q )
{
  int i, j, k;
  float tr, s;
  int   nxt[3] = { Y, Z, X};
  tr = mat[0][0] + mat[1][1] + mat[2][2];
  if ( tr > 0.0) {
    s  = fsqrt( tr + 1.0);
    q[W] = s * 0.5;
    s = 0.5/s;
    q[X] = ( mat[1][2] - mat[2][1] ) * s;
    q[Y] = ( mat[2][0] - mat[0][2] ) * s;
    q[Z] = ( mat[0][1] - mat[1][0] ) * s;
  }
  else {
    i = X;
    if ( mat[Y][Y] > mat[X][X]) i = Y;
    if ( mat[Z][Z] > mat[i][i] ) i = Z;
    j = nxt[i];  k = nxt[j];
    s = fsqrt((mat[i][i] - (mat[j][j]+mat[k][k])) + 1.0);
    if ( i == X) {
      q[X] = s * 0.5;
      s = 0.5/s;
      q[W] = ( mat[1][2] - mat[2][1]) * s;
      q[Y] = ( mat[0][1] + mat[1][0]) * s ;
      q[Z] = ( mat[0][2] + mat[2][0]) * s;
    }
    else if ( i == Y ) {
      q[Y] = s * 0.5;
      s = 0.5/s;
      q[W] = ( mat[2][0] - mat[0][2]) * s;
      q[Z] = ( mat[1][2] + mat[2][1]) * s;
      q[X] = ( mat[1][0] + mat[0][1]) * s;
    }
    else {
      q[Z] = s * 0.5;
      s = 0.5/s;
      q[W] = ( mat[0][1] - mat[1][0]) * s;
      q[X] = ( mat[2][0] + mat[0][2]) * s;
      q[Y] = ( mat[2][1] + mat[1][2]) * s;
    }
  }
}

/***
   Compute the inbetweens for two keyframes ( input1 & input2 ) for "steps"
   inbetweens. Linear interpolation will be performed on the scaling and translational
   vectors as well as the shear parameters. Spherical linear interpolation will be
   performed on the quaternions. After all interpolations are completed the components
```

```
    of each inbetween is combined to a respective 4x4 matrix used in the animation.
***/
void Interpolate(Tmat m1, Vec scale1, Vec shear1, Quat q1,/* input 1 */
            Tmat m2, Vec scale2, Vec shear2, Quat q2,/* input 2*/
            Tmat *matrix, int steps) /* output */
{
   int i, j;
   float  delta, trans[3], scl[3], shear[3], qt[4], t;
   delta = 1.0/(float)steps;
   t = delta;
   for ( i=0; i< steps; i++) {
      for ( j=0; j<3; j++)
         trans[j] = linear( m1[3][j], m2[3][j], t);
      for (j=0; j<3; j++) {
         scl[j] = linear( scale1[j], scale2[j], t);
         shear[j] = linear(shear1[j], shear2[j], t);
      }
      slerp( q1,q2,t, qt);
      QuatToMatrix( qt, matrix[i]);
      /*
          Combine translation, scaling, shear, and rotation to one matrix
      */
      for (j=0; j<3; j++) {
         matrix[i][3][j] = trans[j];
         matrix[i][2][j] = shear[1]*scl[2]*matrix[i][0][j]+
                   shear[2]*scl[2]*matrix[i][1][j]+
                   scl[2]*matrix[i][2][j];
         matrix[i][1][j] = shear[0]*scl[1]*matrix[i][0][j]+
                   scl[1]*matrix[i][1][j];
         matrix[i][0][j] *= scl[0];
      }
      t += delta;
   }
}

/***
    Perform the spherical linear interpolation for the quaternion keyframes to produce
    the quaternion inbetweens. P and q are input quaternions, t is the distance from
    p( between 0 and 1), qt is the interpolated quaternion.
***/
void slerp( Quat p, Quat q, float t, Quat qt)
{
   float  omega, cosom, sinom, sclp, sclq;
   int    i;
   float  temp1, temp2, tempq1[4], tempq2[4];
   /*
      Determine the shortest distance (arclength) on the sphere, use either +q or -q
   */
   temp1 = temp2 = 0.0;
   for ( i=0; i<4; i++) {
      tempq1[i] = p[i] - q[i];
      tempq2[i] = p[i] + q[i];
      temp1 = temp1+(tempq1[i] * tempq1[i]);
      temp2 = temp2+(tempq2[i] * tempq2[i]);
```

```
  }
  if ( temp2 < temp1 ) {
     for (i=0; i<4; i++) q[i] = -q[i];
  }
  cosom = p[X]*q[X] + p[Y]*q[Y] + p[Z]*q[Z] + p[W]*q[W];
  if ( (1.0+cosom) > EPSILON) {
     if ((1.0 - cosom) > EPSILON) {
        omega = facos(cosom);
        sinom = fsin(omega);
        sclp = fsin((1.0 -t)*omega)/sinom;
        sclq = fsin( t*omega)/sinom;
     }
     else {
        sclp = 1.0 -t;
        sclq = t;
     }
     for ( i=0; i<4; i++) qt[i] = sclp*p[i] + sclq*q[i];
  }
  else {
     qt[X] = -p[Y]; qt[Y] = p[X];
     qt[Z] = -p[W]; qt[W] = p[Z];
     sclp = fsin((1.0-t)*HALFPI);
     sclq = fsin(t*HALFPI);
     for ( i=0; i<4; i++) qt[i] = sclp*p[i] + sclq * qt[i];
  }
}

/***
    Linear interpolation for the translation, scale, and shear parameters.
***/
float linear( float p, float q, float t)
{
  return ( p+(q-p)*t);
}

/***
    Convert the quaternion inbetweens back to the required rotational matrix
***/
void QuatToMatrix( Quat q, Tmat matrix)
{
  float s, xs, wx, xx, yy, ys, zs, wy, wz, xy, xz, yz, zz;
  s = 2.0/(q[X]*q[X]+q[Y]*q[Y]+q[Z]* q[Z]+q[W]*q[W]);
  xs = q[X]*s; ys = q[Y]*s; zs = q[Z]*s;
  wx = q[W]*xs;  wy = q[W]*ys; wz = q[W]*zs;
  xx = q[X]*xs;  xy = q[X]*ys; xz = q[X]*zs;
  yy = q[Y]*ys;  yz = q[Y]*zs; zz = q[Z]*zs;

  matrix[0][0] = 1.0 - (yy + zz );
  matrix[0][1] = xy + wz; matrix[0][2] = xz - wy;

  matrix[1][0] = xy - wz; matrix[1][1] = 1.0 - (xx+zz);
  matrix[1][2] = yz + wx;

  matrix[2][0] = xz + wy; matrix[2][1] = yz - wx;
```

```
    matrix[2][2] = 1.0 - (xx + yy);

    matrix[0][3] = matrix[1][3]=matrix[2][3]=
    matrix[3][0] = matrix[3][1]=matrix[3][2]=0.0;
    matrix[3][3] = 1.0;
}
```