COMPUTER ANIMATION: 3-D MOTION SPECIFICATION AND CONTROL

SIGGRAPH 1987 Tutorial

David Sturman
Julian Gomez
Roger Gould
Glenn McQueen
Jane Wilhelms

COURSE #10

TABLE OF CONTENTS

Computer Animation: 3D Motion Specification and Control
SIGGRAPH 1987

COURSE INTRODUCTION
Computer Animation: 3D Motion Specification and Control
SIGGRAPH 1987

David J. Sturman
NYIT Computer Graphics Lab

Computer animation is a fast growing field. In the past ten years improvements in computer speed, size, and cost have made animation by computer feasible for many new applications. Computer animation is now widely used in industry, science, manufacturing, entertainment, advertising, and education. Of the various aspects of computer animation this tutorial will focus on one--motion control. Motion control involves the translation of an idea for a motion or action into it's actualization in a sequence of animation. The ease and accuracy of translation are the criteria with which to measure the quality of a motion control system.

The tutorial is designed for people interested in writing their own motion control systems and for those interested in investigating existing motion control systems.

The five speakers have been involved in the computer graphics field for a number of years. **David Sturman** is a senior research scientist at the New York Institute of Technology's Computer Graphics Lab and is an author of much of the Lab's animation software. He is a co-author of *em* a parameterized keyframe animation system. **Roger Gould** is an animator for Pacific Data Images, one of the world's leading computer animation production houses. He received his BA from Brown University in 1984. He was the coordinator in the production of "A Comic Zoom", a short animation that appeared in SIGGRAPH 1985 and the 1986 film "Animation Celebration". **Dr. Julian Gomez** received his doctorate from The Ohio State University where he authored a publicly available computer animation system, *Twixt*. *Twixt* has been used for a number of animations shown at SIGGRAPH in the past few years and has been the primary computer animation tool used by Cranston-Csuri Productions and the OSU Computer Graphics Research Group. Julian is now with RIACS engaged in research on chaotic attracters and graphic aerodynamic simulators for NASA. **Dr. Jane Wilhelms** teaches computer graphics and animation at the University of California, Santa Cruz and is the author of *Deva*, a computer animation system that uses dynamic analysis to model the motion of jointed bodies. **Glenn McQueen** received his education as a traditional animator at Sheridan College in Toronto and has been a computer animator at the NYIT Computer Graphics Lab for the past 3 years.

Four of the lectures have contributed an original paper to the course notes. In these papers they discuss their own area of expertise as well as comment on the current state of computer animation. Although some concepts are repeated in several papers, each author writes from a unique point of view with different emphasis and meaning.

In addition, several key papers by these and other authors are reprinted. The papers go into specific detail about different techniques of computer aided motion control. The authors were kind enough to grant permission for the reprinting of their work. They include William Armstrong, Glenn Entis, Michael Girard, Patrick Hanrahan, Dick Lundin, Nadia Magnenat-Thalmann, Craig Reynolds, and Ken Shoemake.

In the course itself, the lecturers will discuss the various forms of motion control currently in use. Examples of these techniques will be shown in computer animations from recent years. In addition, the role of the animator, vs. the computer scientist will be discussed with attention to the elements of a computer animation system that are most supportive of the different types of animation and animator. Towards the end of the session there will be time for questions and discussion of particular techniques and computer animation in general.

In preparing for this course the lecturers found themselves asking, "What (or who) is a computer animator?", and, "What constitutes a computer animation system?" They found that answers were numerous, but that none were complete. They also found that the act of asking the questions was in itself valuable insofar as it continually gave perspective and meaning to the more detailed issues. In discussing various computer animation techniques and computer animation systems, the lectures will keep these questions open. They invite the listener (and reader) to keep these questions in mind as well, and, after the course, to ask them of themselves as they evaluate or design new animation systems.

A Discussion on the Development of Motion Control Systems

David Sturman
Computer Graphics Laboratory
New York Institute of Technology
Old Westbury, NY 11568
May, 1986

Introduction

Animation is a process to represent change over time, be it the movement of a 'flying logo', the bending of a robot limb, the contortions of a cartoon alley cat, a near-instantaneous chemical process, or the life-cycle of a star. Animation literally means "to bring to life". It means to take something still and give it vital signs--to make it move.

For the most part, computer animation involves the simulation of motion through the rapid display of successive images. Each image represents a sequential moment in time of an action. When these images are displayed fast enough, the human eye interprets them as continuous motion. When generating animation we are really creating single frames that, when viewed sequentially, appear continuous. We are all familiar with this phenomenon.

Motion control is the specification of the frame to frame changes in an animation that create the illusion of action. Motion control for computer animation has been studied seriously for some twenty years [Magnenat-Thalmann 85]. In that time there has been developed a wide diversity of computer systems for motion specification. Each system attacks a different issue and thus has different strengths and satisfies different needs. No one system has been able to handle the whole task of motion control. This paper discusses various types of systems in terms of the basis for their development and characteristics of their typical users.

Users of Computer Animation

Computer technology has been applied to animation in many ways. The entertainment industry has made heavy use of it in advertising and 'Saturday morning' cartoon generation. In advertising, computer graphics is used to get a 'technical' or 'space-age' look to an animation. In cartooning, computers are used to speed up the process of generating the individual frames of the cartoon. Artists have also used computer animation as a new exploratory art form.

On the other end of the spectrum is the scientific community. Engineers use computer animation for simulating otherwise unobservable or prohibitively expensive dynamics such as fluid flow across an airplane wing or ship's propeller. Manufacturers use computer animation for simulating manufacturing processes, and experimenting with alternative designs before building expensive equipment. The field of robotics has been especially active. Here motion control is used to program industrial robots. The techniques that have been developed are of use in many other applications and the wealth of robotics literature is a basis for many of the recent developments in animation systems.

Biochemists use computer animation to simulate minute chemical processes, the bonding of proteins, the winding of DNA. The visualization of these processes through computer animation is bringing new understanding to their field.

Perhaps the heaviest users of computer animation are the military, aviation, and shipping industries. They make daily use of simulators to train pilots, tank crews, ship captains and operators of other vehicles which are expensive to run. By using simulators, they can train a greater number of people in nearly realistic situations at a significantly reduced cost. In addition, hazardous situations can be simulated with no risk to the participants in training.

Each of these applications has different requirements for a computer animation system. A system designed for one use is not often useful for another. On the other hand, systems often incorporate techniques developed for other applications.

2D Computer Animation

One of the first ideas for the use of computers in animation was to speed up the traditionally laborious process of cartoon animation. Traditionally, cartoon animation is done by hand. Each frame of the animation is drawn by an artist, skilled in drawing sequences of images that come to life. Animators analyze real motion either directly or from film, observing the different components and characteristics of particular actions. Animators understand that the perception of a motion is often different than the motion itself. So, they use 'tricks' to lead the eye and enhance the action. These tricks generally employ non-realistic effects that emphasize motions and give greater life to characters. Objects are squashed, stretched, twisted, enlarged, and otherwise deformed to exaggerate and emphasize their action. These are techniques learned through years of trial and error. Animators acquire a repertoire of motions that they use repeatedly in their work. The pacing and motion of walking, jumping, running, skipping, flying, cloth movement, emotional expression, mouth movement, speech and a host of other actions are all basic knowledge to the traditional animator.

A laborious part of hand animation is the generation of in-between frames. To create an animation, skilled artists draw 'key' frames. These frames represent the extremes of a motion or a critical juncture in an animation. Keys set the general outline and pacing of an animation, and are generally three to five frames apart. In a thirty second sequence of animation there are 720 frames (at film rate of 24 frames/sec) of which one hundred or more may be key frames. The in-between frames ('in between' the key frames) are basically slight modifications of the key frames. These can be drawn (laboriously) by less experienced animators (who can be paid less). To maintain consistency, the "in-betweeners" must imitate the style of the original animator. A good in-betweener is a good imitator. (Additional people do the coloring, backgrounds, camera work, etc.). In this way, an entire feature length film can be made based on the art work of just a few 'key' animators.

When animators turned towards computers it was to reduce the labor and tedium inherent in the animation process. Systems were developed to generate the 'in-between' frames from keys pencilled by an animator [Burtnyk 76, Catmull 78]. Typically the animator draws key frames, carefully drawing each line in a key frame to correspond to a line in the previous key frame. The computer then interpolates the lines, on a point-to-point basis, generating each frame between the hand-drawn keys. This is strictly a two dimensional process, relying on the animator's skill to generate the correct overall motion. The main goal is to speed up the animation process by automating in-betweening. The computer lends a certain look to the animation, but the creativity and design are still determined by the animator.

3D Computer Animation

With the advent of 3-dimensional modelling, animating and rendering, computer graphics differed enough from traditional 2D animation to be a new form of expression. A modeller describes to the computer the detailed environment and models (characters) of a sequence. The description includes lighting, color, shape and surface properties of the models, camera lens characteristics, etc. An animator (often the same person as the modeller) describes the actions that the models, camera and environment perform over time. The computer rendering system then automatically draws each frame of the animation with the correct shading, positioning, and perspective for the scene.

The skills required for 3D computer animation are different than for traditional 2D animation, although they share some common ground. Drawing (drafting) is not as crucial since the computer provides the correct outlines and perspectives on rendering. The initial pacing of motion is not as crucial since, once established, motions can be sped up or slowed down with minimal effort. Nevertheless, the artistic eye traditional animators have for motion and for representing motion lends a very creative aspect to their 3D computer animations. Although their actual drafting skills may not be necessary for 3D animation, their skill in composing and pacing animations are invaluable. Unfortunately, nearly all currently available 3D computer animation systems require a fair degree of computer literacy to generate any meaningful work, restricting their use by traditional animators.

3D Keyframe Systems

At first 3-D systems were developed as logical extensions of 2-D keyframe systems. They used the simple idea of allowing the animator to interactively position a model on a screen, specify a frame number, position the model again, specify another frame number, interpolate, and view the resulting animation. Many hardware devices can perform 3D transformations and display jointed rigid body models in real-time (1/30th sec.) and so interactive systems using those devices give immediate visual feedback of the effects of an animator's manipulations. A simple conceptual basis combined with immediate visual feedback makes keyframed systems the easiest for computer-novice animators to use and the most common type of animation system in use today.

Early keyframe systems used rigid body parts, linked at movable joints [Williams 82]. Position control consisted of rotations and translations about those joints. Many systems allowed scaling of parts, and animation of the eyepoint and perspective transform. Several kinds of keyframe interpolation were usually provided such as linear, cubic spline, etc. allowing the animator a measure of control over the in-betweening.

The number of ways a model can move increases dramatically as the model becomes more complex. The human body, for instance, contains over 200 degrees of freedom. Animators working with early keyframe systems recognized the need for a higher level of control if they were to do more complex animations.

Parameterized Systems

Parameterized systems present a slightly higher level of control over simple keyframe systems. They utilize keyframe interpolation, but the nature of the data stored at the keyframes is different. In parameterized systems the modeling of a character is more closely linked to its animation. The way in which an object can move is specified in the modelling stage rather than assumed, as is the case with simple keyframe systems. Degrees of freedom, coordination between parts, limits of motion, etc. are built into models through the use of modifiable parameters. The animation system provides the animator control over the parameter values which in turn control the positioning of the model. The parameters can also describe the geometry of a model to allow shape changes in the animation. Parameter values are keyed and interpolated as in a simple keyframe system. One such system, *em*, has been developed at NYIT [Hanrahan 85].

Parameter driven animation systems require a good deal more computer training and experience than simple keyframe systems. However, given the extra computer expertise, an animator has more control over an animation than with the direct keyframe approach.

Programming Animation

Often an animator requires a special effect, or a specific motion not handled by a general purpose animation system. In this case the motion has to be programmed directly. This happens mostly in the scientific community but occurs more often than desired in the entertainment industry. Programming usually is done in a common computer language like LISP, C, Pascal or Fortran. Occasionally the language has some extensions to support graphics or animation. Using a programming language gives the animator complete control over the animation; however, he may not see the resulting motion until the program is complete and the full animation is rendered. If the motion is not correct, the program has to be modified, recompiled, re-executed, and the animation re-rendered. This may take several minutes to several hours. An alternative is to provide a programming interface to a general purpose animation system. At NYIT we have a programmer's interface that allows programs to manipulate the animation database used by our keyframe animation system *bbop*. An important part of our animators' tools are special purpose programs that manipulate this kind of database [Lundin 84]. The database can be reused by *bbop* at any time for motion preview and interactive modification. In fact *bbop* itself can be considered an interactive tool for editing animation databases.

The direct programming approach is, perhaps, the hardest way to animate for a user with limited computer experience (animators, biologists, physicians). The quality of the motion is closely tied with the ability of the programmer to translate a concept into an algorithm and implement it. Programs are often special purpose and not used more than once or twice. One of the reasons for this is that it often takes as much time to write a program the first time as to later adapt it for general use or to teach others to use it. The most common users are those with computer expertise who can program their own animations or are employed to program the animations of others.

Scripting Systems

Scripting systems were developed in an effort to provide a method of animating that was more flexible (and thus more powerful) than a keyframe approach and not as difficult to work with as a full blown programming language. Scripted animation uses a program-like script to 'model' the animation of a scene. The script usually describes both the models and animation of a sequence. When describing his scripting animation system ASAS, Craig Reynolds refers to it as, "a notation for animated graphics" [Reynolds 82]. As a notation it can be read, edited, and modified on any general purpose computer system. Only when actually rendering images is any special hardware required.

The animator using a scripting system designs and writes the animation script, and then runs it through the system once for each rendered frame. It may take several minutes (or longer) to generate a complex sequence for playback, thus feedback can be slow. The advantage of a scripting system is that it provides the flexibility of programming while still supporting animation specific capabilities. Libraries of motions and models can be generated and reused making the animation process simpler as time goes on. In the case of LISP-based systems, the scripting language is extensible so that new animation functions and primitives can be added to the system. Scripting languages are usually easier to learn than general-purpose programming languages and thus easier for people who are not experienced programmers.

An early scripting system, GRAMPS [O'Donnell 81], takes advantage of a fast, interactive graphics device to allow real-time manipulation of parameters. In this way the animator can interactively set values which are then used in the interpretation of the script.

The Scientific Approach

Simulation

For the most part, animation for entertainment is not concerned with realistic simulation. The goal is communication: if a sequence communicates more by ignoring reality, e.g. an accordion shaped cat walking away from the crushing blow of 20 tons, then animators have no hesitation in using it. Not so for the scientific and engineering community. Realism is important. Looks are not. When a molecule bonds with another there is no squishing or squashing to emphasize the fact. In simulation, the purpose is to describe what happens, whatever happens.

Most simulations are programmed directly. When the description of the motion is clear and well defined, programming is a natural way to proceed. A few general purpose animation systems are used for simulation, but the simulation and entertainment communities have different enough goals that systems well suited for one may not be very helpful to the other. Even so, some of the basic technology is shared and many techniques developed in one field are used extensively in the other.

Robotic Origins

Many techniques for 3-D motion systems come from the robotics field where the emphasis is on the control of robot arms and manipulators.

The robotics industry has developed an extensive literature in the problems of programmed robot control. Some of the major topics dealing with motion control are forward and inverse kinematics, motion trajectories, dynamics, collision detection, and path planning [Paul 81, Proc. IEEE Robotics 85]. Most robotics systems are language oriented systems in which the engineer (animator) writes a series of instructions to program the robot [Paul 77]. Some systems teach robots by example [Puma 84]. These systems allow the operator to position a robot arm at certain points in a trajectory. The robot system then remembers these points and can calculate a smooth path of motion through them.

In some ways the problems for computer animation are different from those of robotics. Robotics is not concerned with problems of expressive communication, or non-mechanical simulation. However, the underlying techniques for motion simulation are the same, and a study of robotics literature is important for anyone developing a computer animation system.

Inverse Kinematics

Inverse kinematics involves the determination of joint rotations and part lengths that result in precise motion, placement, and orientation of an end node (e.g. a robot hand). This is useful, for instance, when you know that you want a button pressed, and you don't care how the robot arm gets there, as long as it does. Or in the case of walking where you want the foot to remain planted on the floor while the knee bends appropriately to the motion of the hip.

When working with an animation system that implements inverse kinematics, the animator specifies discrete positions and motions for end parts. The system then computes the necessary joint angles and orientations for other parts of the body to put the specified parts in the desired positions and through the desired motions. Inverse kinematics is most easily applied to bodies of few linkages. It works well for walking and for arm/hand positioning [Girard 85]. As the number of linkages increases, the inverse kinematic solutions to a particular position become numerous and complicated. Computation slows down considerably and more information must be supplied by the animator.

For example, it is comparatively simple to determine how much to bend an elbow and twist an upper arm to put a hand into a mailbox. When you bring into play the rotation of the shoulder, the problem becomes a little more difficult. At this point, you have to specify which takes precedence: the twist of the upper arm or the rotation of the shoulder. Choices may be based upon which is more energy efficient, communicates better, or is more natural. As you add degrees of freedom, especially ones that create redundant degrees of freedom, the problem becomes more complicated since there may be many energy efficient, or natural solutions to a single positioning. The criteria for selecting must be made more specific, thus more animator intervention. When criteria are natural, humorous, ponderous, etc. the specification itself becomes difficult. (What does natural mean?)

One of the drawbacks of inverse kinematics is mentioned above, namely that it is not simple to use for complex linkages. Another drawback, perhaps more influential, is that inverse kinematics is not applicable to all aspects of animation and must be incorporated in an animation system along with more general techniques. It has been used mainly in the industrial sector for robotics. A few researchers have successfully experimented with it for use in production animation, and produced impressive animations, but none have been embodied in a general purpose animation system [NYIT Demo 84, OSU Demo 84].

Goal Directed Systems

Goal directed systems arise out of the concept of director/actor. The director says, "walk over there", or "pick the wrench up from the table", and the actor obeys. One of the first goal directed systems was SHRDLU developed by Winograd in 1971 as part of a landmark thesis in artificial intelligence [Winograd 72]. It dealt with a very limited block-world yet was very effective within this world. The user could direct the system to put a red block on a blue block and the system would figure out where the two blocks were and what motions to perform to get the one on top of the other. David Zeltzer has developed a goal directed system based on the idea that objects can be given 'motor skills' and then directed to use those skills [Zeltzer 82, Zeltzer 84]. Research in goal directed systems has also been going on at the University of Pennsylvania [Korein 82].

The main effort of using this type of system is 'teaching' objects required skills and giving them sufficient knowledge to act consistently within their environment (for example, avoiding obstacles or walking over uneven terrain). The ease of use of these systems lies entirely with the interface presented to the animator. Their limitation lies in the mass of background information needed in order to direct a character through a variety of motions in a non-trivial environment. However, once a skill is taught, it can be used repeatedly with little work. This is a system that gets more powerful with extended use.

Dynamics

Although inverse kinematics entails solving the body position for a desired effect it does not take into account the mass and inertia of the body in motion. Thus animations produced by these systems often have an unrealistic appearance. The objects do not seem to have weight or mass, and thus speeds of movement are inaccurately represented. Dynamics takes into account mass and inertia as well as the various forces acting on the body. Animations come out realistically at the expense of much (slow) computation per frame. Characters move correctly and appear to have weight and substance.

General purpose animation systems incorporating dynamics are still in the experimental stage. For the most part, they have been used in robotics to simulate robot arm masses in motion. They require some programming to set up a model with the correct characteristics and then to apply the correct forces. Once that is complete, the system works out the rest for itself. Recently, Jane Wilhelms developed *Deva*, an interactive system for setting up and previewing the dynamic motion of articulated bodies [Wilhelms 85].

Dynamics may be the best way to achieve realistic motion for simulations. Combined with goal direction, it could provide an excellent basis for a general purpose animation system. Special effects involving unrealistic motions will still be needed so the ability to override the dynamic control will be necessary.

Other Techniques

There are an infinite variety of motions. The techniques described above only address a few of them. Other techniques have been developed to animate specific behaviors. These include particle systems [Reeves 83], systems that act on amorphous forms [Kawaguchi 85], group animation [Amkraut 85], and others. These systems handle special cases of animation and are of limited use except when integrated into a more general purpose animation system. They often require more skill in programming than in animation yet hold an important place in the overall scheme of animation.

Discussion

As indicated, there are many techniques and systems used for motion control for computer animation. Each one answers a particular need or caters to a particular class of user. No one system answers all needs, yet each system has an appropriate place in the computer animation tool bag. This is a natural state in the evolution of any discipline. As time goes on these systems will be integrated so that an animator has many of these techniques available at once.

The recent trend in animation systems has been towards higher levels of control [Korein 82, Zeltzer 85, Wilhelms 85]. Higher levels of control are always based on lower levels of assumption. To make a dynamics simulation perform correctly requires the pre-specification of masses, forces, links, etc. The higher the level of control, the more pre-specification is required. Some systems have a default set of assumptions to simplify the animator's set-up. Nonetheless, this still requires an initial specification of the default set-up. Sometimes non-animators find a system easy to use but have difficulty with the initial set-up. For instance, animators at NYIT find that the initial set up for the parameter system, *em*, is relatively difficult, but that *em* affords better control over the model and animation than the conventional keyframe system, *bbop*. Experience has shown, though, that *bbop* offers sufficient control for most animations and, because of the easier set-up, is more likely to be used. Occasionally, animators find that neither system fits their needs and so program the motion themselves. In such cases, they often bring the completed motion back to the general purpose system to do touch-up work or to coordinate it with other action in the animation. Thus, when providing higher levels of control, it is important that the increased level of control not require a difficult set-up, and that the animator still has access to all lower levels of motion control.

One difficulty in the design of animation systems is that systems are designed by computer scientists based on their own understanding of the needs of animators, or are designed by computer scientists who want to do animation themselves. Rarely (if ever) are systems designed and implemented by animators. Industrial and scientific animators often have a specific idea or set of rules and algorithms from which a motion control system can be designed. This can help in getting the animation capabilities right, but even so, the resulting system is often easier to use for the computer scientist than the animator.

Animators usually want something to happen, an effect, such as a ball bouncing and hitting a wall, or a shock wave crossing an airfoil. They do not want to have to write down (or know) the equations for the motion of bodies in the influence of gravitational forces, or the formulas for fluid flow. They may just want to control the weight of the ball or the shape of the airfoil. But, they find that they cannot get the effects they desire without programming. In many computer animation production houses computer programmers work alongside animators, writing special purpose programs and using more complicated animation systems to provide effects that the animator can visualize but not implement. The scientific and engineering communities are analogous. A solution may be to provide systems that do not require programming skills to use. This would open the field to more traditional animators. Then again, perhaps animators should learn to program. Systems may be more powerful and versatile if users can express their ideas algorithmically. Up until now, most animators have found it necessary to learn programming, but the field is still young and has not taken root on any particular ground. These issues will be dealt with slowly as the various users and implementors work together, each building upon the skills and understanding of the other.

References

[Amkraut 85] Amkraut, Susan, Girard, Michael, and Karl, George. Eurythmy. ACM-SIGGRAPH '85 Film and Video Show. July, 1985 A film from the Computer Graphics Research Group, Ohio State University.

[Burtnyk 76] Burtnyk, N., and Wien, M. Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key Animation. *Communications of the ACM* 19(10):564-569, October, 1976.

[Catmull 78] Catmull, E. The Problems of Computer-assisted Animation. In *Computer Graphics:SIGGRAPH '78 Conference Proceedings*, pages 348-353. ACM-SIGGRAPH, August, 1978. A good overview of the problems in computer-assisted 2-D cel animation.

[Girard 85] Girard, Michael, and Maciejewski, Anthony A. Computational Modeling for the Computer Animation of Legged Figures. In *Computer Graphics:SIGGRAPH '85 Conference Proceedings*, pages 263-270. ACM-SIGGRAPH, July, 1985.

[Hanrahan 85] Haranhan, Pat, and Sturman, David. Interactive Animation of Parametric Models. *The Visual Computer:International Journal of Computer Graphics* 1(4):260-266, December, 1985.

[Kawaguchi 85] Kawaguchi, Yoichiro. Growth III:Origin. ACM-SIGGRAPH '85 Film and Video Show. July, 1985 A film produced with the LINKS-1 at Osaka University, and The Art & Science Laboratory, Nippon Electronics College.

[Korein 82] Korein, James U., and Badler, Norman I. Techniques for Generating the Goal-Directed Motion of Articulated Structures. *IEEE Computer Graphics and Applications* 2(9):71-81, November, 1982.

[Lundin 84] Lundin, Richard V. Motion Simulation. In *Nicograph 1984 Conference Proceedings*, pages 2-10. Nicograph, November, 1984.

[Magnenat-Thalmann 85]
Magnenat-Thalmann, Nadia, and Thalmann, Daniel. An Indexed Biblography on Computer Animation. *IEEE Computer Graphics and Applications* 5(7):76-85, July, 1985.

[NYIT Demo 84] New York Institute of Technology. SIGGRAPH 84 Demo Reel. ACM-SIGGRAPH '84 Electronic Theater. July, 1984

[O'Donnell 81] O'Donnell, T.J., and Olson, Arthur J. GRAMPS - A graphical language interpreter for real-time, interactive, three-dimensional picture editing and animation. In *Computer Graphics:SIGGRAPH '81 Conference Proceedings*, pages 133-142. ACM-SIGGRAPH, August, 1981.

[OSU Demo 84] Ohio State University Computer Graphics Research Group. SIGGRAPH 84 Demo Reel. ACM-SIGGRAPH '84 Electronic Theater. July, 1984

[Paul 77] Paul, Richard P. WAVE: A Model-Based Language For Manipulator Control. *The Industrial Robot* 4(1):10-17, March, 1977.

[Paul 81] Paul, Richard P. *Robot Manipulators: Mathematics, Programming, and Control.* The MIT Press, Cambridge, MA, 1981.

(Proc. IEEE Robotics 85)
IEEE. *Proceedings: International Conference on Robotics and Automation*, 1985.St. Louis.

[Puma 84] *Unimate Puma Mark-II Robot 500 Series Equipment and Programming Manual* Unimation, Inc., Danbury, CT, 1984.

[Reeves 83] Reeves, William T. Particle Systems - A Technique for Modeling a Class of Fuzzy Objects. *ACM Transactions on Graphics* 2(2):91-108, April, 1983.

[Reynolds 82] Reynolds, Craig W. Computer Animation with Scripts and Actors. In *Computer Graphics:SIGGRAPH '82 Conference Proceedings*, pages 289-296. ACM-SIGGRAPH, July, 1982.

[Wilhelms 85] Wilhelms, Jane, and Barksy, Brian. Using Dynamic Analysis to Animate Articulated Bodies Such as Humans and Robots. In *Proceedings, Graphics Interface '85*, pages 197-104. May, 1985.

[Williams 82] Williams, Lance. BBOP. In *Course Notes: Seminar on Three-Dimensional Computer Animation.* ACM-SIGGRAPH, July, 1982.

[Winograd 72] Winograd, Terry. *Understanding Natural Language.* Academic Press, 1972. PhD Thesis. Also discussed in many books on Artificial Intelligence.

[Zeltzer 82] Zeltzer, David. Motor Control Techniques for Figure Animation. *IEEE Computer Graphics and Applications* 2(9):53-59, November, 1982.

[Zeltzer 84] Zeltzer, David. *Representation and Control of Three Dimensional Computer Animated Figures.* PhD thesis, The Ohio State University, August, 1984.

[Zeltzer 85] Zeltzer, David. Towards an Integrated View of 3-D Computer Character Animation. In *Proceedings, Graphics Interface '85*, pages 105-115. May, 1985.

Interactive Keyframe Animation of 3-D Articulated Models

David Sturman
Computer Graphics Laboratory
New York Institute of Technology
Old Westbury, NY 11568

Abstract

This paper discusses some of the issues concerned with keyframed computer animation of 3-D articulated models and the problems in designing interactive systems for this type of animation. Examples are taken from the four years of keyframe animation of articulated models done at the NYIT Computer Graphics Lab, and from our recent attempts to refine our original keyframe animation system, *BBOP*. This paper also addresses the importance of the interaction of animators with keyframe animation systems as an element in the design of such systems.

Résumé

Cet article aborde certains des aspects de l'animation par ordinateur traitée par images intermédiaires de modeles 3-D articulés et des problemes de la conception de systemes interactifs appliqués à ce genre d'animation. Des examples décrits sont extraits de nos quatre ans d'expérience en animation par images intermédiaires de modeles articuleś au sein du NYIT Computer Graphics Laboratory et de nos récentes tentatives d'améliorer notre systeme d'animation par images intermédiaires, BBOP. Cet article souligne aussi l'importance de l'interaction des animateurs avec des systemes d'animation par images intermédiaires comme un élement de la conception de tels systemes.

KEYWORDS: Animation, articulated model, keyframe, interactive, user-friendly.

Introduction

Animation for film and video has traditionally been a long and tedious task, with many animators drawing and painting each frame by hand. With the advent of computers, animators turned toward these machines to take the drudgery out of the animation process. Early systems worked with two--dimensional images, automating the tedious inbetweening task. Animators specified the correspondence between lines in successive key frames by tracing them in a fixed order. The computer would generate the frames in between by interpolating corresponding lines from the keyframes [1,2]. This technique has been refined and is still very popular in modern animation systems like the TWEEN system produced by CGL, Inc. [3]

As three dimensional animation became possible with faster machines and better display hardware, keyframing was adapted to 3-D animation. Instead of interpolating corresponding 2-D line segments, 3-D animation systems interpolate transformations at joints in a three-dimensionally represented model. Key frames consist not of 2-D images, but of 3-D positions of a model.

Because the models were represented in 3-space and projected on the image plane by the computer, these systems tend to produce more realistic-looking images, than those produced by an animator who approximates a 3-D representation with a 2-D drawing.

This paper discusses some of the techniques and problems in the design of 3-D keyframe animation systems, stressing the importance of animator interaction. It draws heavily on the NYIT Graphics Lab's keyframe animation system BBOP[4,5], and a more recent NYIT animation system, EM[6].

Models

The information stored in a model is an important aspect of any animation system. One simple way to define models is as a set of rigid objects jointed at nodes, organized hierarchically into an articulated body. At each node or joint, a 3-D transformation matrix controls the position of the portion of the body below that joint. Transformation matrices are nested in accordance with the body structure. The position of the model at any one instant is determined solely by the transformation matrices. The only intelligence contained in the model is the topology of the body parts and the degrees of freedom at each joint. Alone, the model is a static entity. To make the model move, the animator uses the animation system to control the 3-D transformation values at each joint. The "rigid object" stipulation allows scaling of the body parts (using the joint matrices) but not flexing or changing their basic geometries. These are the types of models used by the animation systems BBOP and GRAMPS[7]. The particular model structure was motivated by the Evans & Sutherland Multi-Picture System (MPS), on which the systems are based. The MPS manages nested transformations easily: performing real-time transformation, clipping, and display of lines.

Like BBOP, EM uses models constructed of parts connected at joint nodes. However, EM uses a geometric modeling language which allows parametric control over the transformations at each joint and over the geometry of the individual body parts. The set of parameters define the model's final position, shape, and characteristics. Parameters can be constrained and coordinated with respect to constants or other parameters in order to give the model intelligence in the way it moves. For instance, the motion of a ball can be dependent on the slope of the floor it rolls across, or swinging arms can be made to swing opposite to each other.

Scripted systems [8,9,10,11] use procedural models [12], which embed the possible motions of the model in the model description itself, leaving some parameters for external control.

Regardless of the implementation of its models, an animation system should allow animators to easily modify model structure and movement. Interaction and visual feedback are important. Systems like BBOP, EM, and GRAMPS are significant in this regard because the animator can immediately view his changes.

Positioning models

Because keyframed animation is based on key "still" frames, the method of creating these frames is a vital component of the system. The animator must be able to easily set up all the parameters necessary to define a keyframe. One basic method of positioning a model in a keyframe is to type in a joint identifier, the parameter to be changed at that joint, and the value for that parameter. Although functional, this method is not easy to use.

In BBOP the animator selects a joint using a joystick to traverse the transformation tree of the model. With a set of function keys, the animator specifies that he wants to modify the translation, rotation, or scaling parameters at that joint. Each parameter set has three elements, one for each of the x, y, and z axes. Using a three-axis joystick, the animator can modify these three values and watch the model move on the screen. BBOP provides no constraints or rules about moving the model, except that it follows the x, y, z movements of the joystick. When manipulating a human body model, for instance, limbs can disconnect from the body structure and joints can be bent at unrealistic angles, even causing a limb to enter the body itself. At each joint, the interaction is the same: the x, y, and z, translation, rotation, and scaling parameters are controlled by the three outputs of the joystick. This makes the system simple to use, and an inexperienced person can learn to manipulate a model in just a few minutes.

GRAMPS, which has a similar interactive input method, takes a step beyond BBOP by allowing several devices for input. By means of simple functional assignment statements, values from eight dials, a data tablet, and a joystick can be used to modify the model's parameters. For instance, the animator can assign the inverse value of a dial to the rotation of a character and set bounds on the values the rotation can assume. EM builds on BBOP and

GRAMPS, allowing input modes to be defined dynamically with complicated dependency expressions including other parameters and multiple input devices.

We noticed an interesting phenomena in systems which have user configurable inputs. In BBOP, the interaction modes are predefined and remain the same from joint to joint and model to model. An animator can sit down with a new model and immediately begin to manipulate it. In EM, the interaction modes can be configured specific to each parameter and joint in the model's tree structure, and by each user. Each model and each user can have different input modes. Thus, it is difficult to sit down with a new model and animate it right away. It takes time for the animator to get used to the model's control characteristics. Once familiar with a model's movement, however, configurable inputs have proven to be very effective. Especially useful has been the ability to interactively control parameters of several joints of the model at the same time. For instance, the animator can rotate the shoulder, wrist, and waist of a model simultaneously. This was not possible with BBOP.

Modeling camera movement is also an important part of an animation system. There are many ways to move the camera interactively. We first must define the coordinate axes in which the camera can be moved. We define the pivoting of a camera on its own axes, i.e. tilting, panning, and rolling the camera, as rotation in the camera's local coordinate system. Camera movement around an external center point is movement in a global coordinate system. This is exemplified by a camera mounted on a crane that moves around the space of the "animation studio." In BBOP, the center of global camera movements is always the center of world space and cannot be changed. Movement along arbitrary coordinate axes would be an important enhancement to the camera model. This would aid in tracking a movement of the model or moving the camera along a particular trajectory while maintaining a constant object of interest.

The second aspect of camera movement is how the camera moves relative to the model or scene. One approach to interactive camera movement is to have the joystick (or other interactive device) operate as if it were attached to the camera. This is the local or "airplane" method of camera control, because the joystick models the pitch, yaw, and roll of an airplane joystick. Pushing the joystick forward causes the camera to tilt down; pulling the stick to the side causes the camera to tilt to that side. The second method is to model the input as if the animator were controlling the world. In this "global" mode, joystick control creates just the opposite effect of the "airplane" method. Pushing the joystick forward causes the camera to tilt up (or apparently, the world tilt to down, away from the eye). The two methods are computationally equal since moving the camera to the left is indistinguishable from moving the world to the right. Different people favor different approaches; however, the majority prefer the "move the world" approach because the picture they see moves in the same direction as the joystick. BBOP and EM also can simulate multiple cameras, enabling an animator to display the same animation as seen from several viewpoints.

In addition to camera movement, there is local and global movement at the joints of the model. That is to say, transformations at a joint can take place in the coordinate system local to the joint or in the coordinate system of the next higher, or "parent," joint. (In an ideal implementation the animator would be able to effect transformations in any coordinate system. This would aid, for instance, in making a figure walk so that it rotates over the balls of the feet, not around the center of the body.) Because BBOP has just one interaction mode – joystick x, y, z controlling model parameters x, y, z, – the joystick's motion often has little physical relation to the motion of the model on the screen. To get around this problem, the animator can display the local coordinate axes as they move with the current joint. EM fosters more natural interaction because the animator designs the input modes, adapting each to a particular manipulation of the model. An example of this is to set the tablet x/y to be the model x/z position on a floor, and the joystick x, y, z to the model's rotation around its z, x, y axes respectively. Pushing the joystick away from you would make the model tilt away from you.

The most natural input method perhaps would allow the animator to point directly to a joint and "drag" it around on the screen. Thus the animator would manipulate the model by pushing and pulling it into position. There are, however, implementation problems in trying to manage 3-D control from a 2-D view. For instance, determining the coefficient of movement along the axis perpendicular to the screen can be quite complicated. A typical solution would be to set that "z" coefficient to zero, but a more sophisticated solution would be better.

Finally there are non-kinetic parameters such as color, reflectance, elasticity, etc. which the animator may need to control. These qualities may be required if a raster version of the animation is desired, and sometimes are perceptible only in a fully rendered scene. The time it takes to render scenes is usually prohibitive to display the actual property to the user in an interactive fashion. An alternative form of viewing these values is necessary. In EM, the animator can learn the current value of any parameter by typing its name. With color vector devices, part of this problem can be solved by coding various properties with different colors.

Keyframing

A frame in an animation is basically a description of the particular state of the world at a particular instant in time. In a keyframe system, the animator need not describe each frame. Instead he describes a set of "key frames" from which the animation system can interpolate the frames in between. Interpolation of intermediate values can be linear, cubic spline, cosine, etc.

The information stored at a keyframe may vary from a total description of the scene and animation to the value of a single parameter used in the generation of a frame. In scripted systems, the "key" information is more like a cue to a stage performer [9]. The cues tell the models (or actors) to start, stop, and in some cases to modify or switch their behavior to some other pre-programmed mode. In goal-oriented systems, the animator describes a goal state and the model moves towards that state using its knowledge of itself and the world [11,13]. Goal states can be considered the keys with the model itself generating the inbetween movements.

To set up keyframes in BBOP or EM, the animator positions the model on the display screen. Then he can record the position of the whole model, a subset of the model, or just the value of a single parameter as a keyed position in a particular numbered frame. In 3-D systems, each parameter has its own set of keys, unlike 2-D systems which key a whole image. In BBOP, every value at every joint for every frame is saved. Those values to be used as keys have a note to that effect. This is a simple but storage-intensive implementation. When an animator specifies a position as a key in EM, the system saves only the values of keyed parameters, the frame number, and interpolation information about the interval between that key and the next. EM has been implemented in a way that eliminates the need to maintain values at the inbetween frames. This scheme is more complicated than BBOP's, but more space-efficient. EM also can save key positions by name, separate from the interpolated sequence of numbered keyframes. Numbered keyframes can contain references to named keys, thus providing a sort of macro facility to the keyframe process. For example, if the same position is needed in multiple keyframes, the position can be described once and then referenced by each key. If the position needs to be changed, modification of the original copy modifies it in the other keys.

An animator often uses the same key many times. Copying keyframes from one position to another at first seems to be conceptually simple, but when looked at carefully a number of issues become apparent. Frequently, an animator copies (or moves) keys forward or backward in a sequence to change the pace or timing of an animation. Moving a keyframe forward to expand a sequence can be done by moving the key to the new location, pushing subsequent keys forward in the process. This expands the particular key interval but also lengthens the entire animation. For modification of the timing of an entire animation, or non-synchronized motions, this is a valid approach. However, moving a key for a motion that is, at some point, synchronized or cued to other motions will push the subsequent keys out of synchronization. Clearly the subsequent keys cannot be moved forward with any integrity unless corresponding keys for the entire animation are also moved forward. Thus, lengthening a keyframe interval for a subset of a model's parameters requires a corresponding contraction of another keyframe interval. The reverse is also true.

In expanding or contracting an interval, an animator may wish to preserve the characteristics of the original motion in the interval. For the most part, interpolating functions take care of that. However, there are cases in which the modified timing of the intervals produces a motion with undesirable characteristics. This is especially true with cubic spline interpolation where keyframe spacing affects the shape of the interpolating curve.

Creating motion cycles is another technique that requires copying keyframes. Repeated motions are common in animation, as in the cycle of a walk or swing. To generate a cycle, an animator can manually copy keyframes of the basic motion to multiple locations, one for each repetition of the cycle. This method is tedious and error-prone. Often the original keys contain information unnecessary to the cycle and need to be trimmed down. Additionally, if the motion is to be repeated many times, the keyframe copy process is unwieldy. Ideally, some form of cycle operator or interpolation should exist in the animation system.

An alternative approach to modifying animation pacing or cycle generation is to manipulate the sequence of frame playback. An animator might draw a curve as a function of time to indicate to the animation system the frame playback sequence he wants. This curve could be saved and later used to control frame playback. A ramp function might be used to produce linear playback of frames at a speed dependent on the slope of the ramp. Cosine curves would create cyclical animation. Acceleration and deceleration could all be described in terms of this function. With further refinement, different parts of a model or animation could become subject to different pacing functions.

As well as being able to key motions, the animation system should make it possible for the animator to key attributes such as color, reflectance, etc. A flexible database is necessary if these attributes, varied in number, are to be added to a model without destroying previous animation.

Using keyframe systems, an animator must manage scores of joints across hundreds of frames; perhaps hundreds of individual keys. Several systems have successfully tackled the problem of giving such control to the animator. One such system, MUTAN[14], divides a model or group of models into tracks – separate entities that can be keyed individually. For each track, the system displays a sort of ruler that spans the range of frames. Tick marks indicate keys and include notations about the action at that key. Animators manipulate the tick marks to change key positions. MUTAN also has a mode which the animator can use to deal with the set of keys at a particular frame. Another system, DIAL[15], employs a specialized notation that the animator edits on a regular alphanumeric terminal. It displays frames horizontally and tracks vertically on the screen. The animator can view parallel tracks at once to facilitate coordination of keys. BBOP and EM have a special motion editor to manipulate keys along a single track (or parameter), and a command that prints the list of key frames for a particular parameter.

Interpolation

The interpolation process has been given little attention in many animation systems. Usually they only support linear and cubic spline interpolation. Some systems allow cosine interpolation and acceleration or deceleration functions. Animators' experiences with BBOP indicate that control of the inbetween frames is very important. There are cases in which the animator only needs linear or cubic spline interpolation, but there are also cases in which a more sophisticated motion is desired. One way to accomplish this is to add more keyframes. With this method, however, the animator quickly gets lost in a forest of keys, and the efficiency of the computer in-betweening is lost. Another solution is to offer a variety of interpolation types which can be selected for the intervals between keyframes. This scheme is better but the interface must give the animator a clear picture of the various interpolation types and the motions they produce. Limiting the animator to typing in keyframes and interpolation information may not be sufficient.

BBOP addresses this problem by providing a motion editor. The animator uses the motion editor to view the values of a parameter across the span of the animation. Key frames are clearly marked along the curve. The animator can add or delete keys and specify one of several functions to interpolate values in individual keyframe intervals. In addition, the animator can hand draw the desired motion between keyframes to achieve unique movement.

With more than one type of interpolation possible, especially across the path of a single parameter, it is important how curves of different interpolation types are joined. To control the continuity of the overall animation, the animator must be able to determine the degree of continuity across keyframes. Also, the interpolant's behavior at key boundaries is an important factor in its behavior between keys. ASAS solves this problem by using piecewise cubic curves with a selectable degree of continuity at the joints. MUTAN lets the animator specify acceleration or deceleration functions at keys. The BBOP motion editor supports an ease-in/ease-out function to match slopes at key boundaries. Using BBOP, an animator usually positions models at key frames, previews the motion generated by the default cubic interpolation of the inbetween frames, and then fine-tunes the movement in the motion editor. The motion editor allows an animator to give characters idiosyncratic motions, limps, jerkyness, and human-like qualities that linear and cubic interpolation would not provide.

Conclusions

The art of 3-D animation goes beyond positioning models, setting keyframes, and interpolating the inbetweens. Although many animations can be made with these methods, a wide range of situations require more. Scripted animation systems provide one set of solutions. They tend to allow a high level of control over an animation, simplifying many types of motion control, and are often used to model algorithmic or functionally-defined motions. These systems are well suited to goal-directed animation and the simulation of mechanical processes. However, scripted systems are not good at producing the idiosyncratic and non-algorithmic "natural" motions that professional animators favor in their productions. In addition, they do not provide immediate feedback – an important element in animation systems geared towards animators. Clearly, some combination of scripted and interactive keyframe animation is desirable.

The combining of the two approaches has been researched over the past year at NYIT, primarily by Pat Hanrahan. The animation system, EM, that has grown out of this research, is more clearly detailed in a paper submitted to the ACM SIGGRAPH '84 conference.

Acknowledgements

BBOP was created by Garland Stern, who developed and inspired many of the ideas in this paper. Kenneth Wesley enhanced BBOP to its current state. The BBOP motion editor was developed by Thaddeus Beier. EM was conceived of by Pat Hanrahan, and implemented with the help of the author. Jacques Stroweis provided the French translation to the abstract. Pat Hanrahan, Paul Heckbert, and Jane Nisselson were extremely helpful in providing constructive review and correction to this paper.

References

(1) Burtnyk, N. and Wein, M., "Computer generated key frame animation." *Journal of SMPTE* 80 (March, 1971): 149-153

(2) Catmull, Edwin, "The problems of computer-assisted animation." *Computer Graphics (SIGGRAPH '78 Proceedings)* 12 (August 1978): 348-353.

(3) *Tween Users Manual.* (New York: CGL Inc., [1983]).

(4) Stern, Garland, "Bbop – a system for 3D keyframe figure animation." *SIGGRAPH '83, Course 7, Introduction to Computer Animation,* July 1983: 240-243..

(5) Stern, Garland, "Bbop – a program for 3-dimensional animation." *Nicograph '83 Proceedings.* Tokyo, Japan, December 1983: 403-404.

(6) Hanrahan, P., and Sturman, D., "Interactive control of parametric models." submitted to *Computer Graphics (SIGGRAPH '84).*

(7) O'Donnell, T. J. and Olson, Arthur J., "GRAMPS – A graphical language interpreter for real-time, interactive, three-dimensional picture editing and animation." *Computer Graphics (SIGGRAPH '81 Proceedings)* 15 (July 1981): 133-142.

(8) Hackathorn, Ronald J., "Anima II: a 3-D color animation system." *Computer Graphics (SIGGRAPH '77 Proceedings)* 11 (July, 1977): 54-64.

(9) Reynolds, Craig W., "Computer animation with scripts and actors." *Computer Graphics (SIGGRAPH '82 Proceedings)* 16 (July 1982): 289-296.

(10) Magnenat-Thalmann, N., and Thalmann, D., "The use of high-level 3-D graphical types in the Mira animation system." *IEEE Computer Graphics and Applications* 3 (December 1983): 9-16.

(11) Zeltzer, David, "Knowledge-based animation." *Proc. ACM SIGGRAPH/SIGART Interdisciplinary Workshop, Motion: Representation and Perception.* Toronto, Canada, April 1983: 187-192.

(12) Newell, Martin E., "The utilization of procedure models in digital image synthesis." Ph.D. dissertation, Department of Computer Science, University of Utah, 1975.

(13) Korein, James U. and Badler, Norman I., "Techniques for generating the goal-directed motion of articulated structures." *IEEE Computer Graphics and Applications* 2 (November 1982): 71-81.

(14) Fortin, D., Lamy, J.F., and Thalmann, D., "A multiple track animator system for motion synchronization." *Proc. ACM SIGGRAPH/SIGART Interdisciplinary Workshop, Motion: Representation and Perception.* Toronto, Canada, April 1983: 180-186.

(15) Feiner, S., Salesin, D., and Banchoff, T., "Dial: a diagrammatic animation language." *IEEE Computer Graphics and Applications* 2 (September 1982): 43-53.

Visual Computer

Interactive animation of parametric models

Pat Hanrahan and
David Sturman

Computer Graphics Laboratory,
New York Institute of Technology,
P.O. Box 170, Old Westbury,
NY 11568, USA

This paper describes a program which allows parametric models of three-dimensional characters and scenes to be interactively controlled for computer animation. The system attempts to span the two most common approaches to animation: language-driven or programmed and visually-driven or interactive. Models are designed in a geometry language which supports vector and matrix arithmetic, transformations and instancing of primitive parts. As a result, constraints and functional dependencies between different parts can be programmed. Control is achieved by parameterizing the model. Subsets of parameters can be connected to different logical input devices, establishing an input mode to control the model's shape. Parameter sets can be stored to form a database of positions. Positions then can be mapped to frames and interpolated to animate the model.

Key words: Animation – Motion control – Three-dimensional graphics – Geometric modeling – Interactive techniques – Parametric models – Articulated figures – Languages

Three-dimensional computer animation draws its power from the partnership between the animator who describes a scene's characters and environment once and the computer which, from this description, is able to synthesize images from different views, adding details such as color, texture, and lighting. Recently great strides have been made in describing and rendering three-dimensional environments (e.g., Smith 1983; Tucker 1984). Computer animation, however, poses additional problems to those of generating static imagery. One has to control changes in the shape of objects, their movement, and their surface properties such as color and reflectance, as well as the methods used to render and compose the individual frames.

There have been two major approaches in the design of animation control systems. The first type of system, exemplified by ANIMA-II (Hackathorn 1977), ANTS (Hackathorn et al. 1981), ASAS (Reynolds 1982), and MIRA-3D (Magnenat-Thalmann and Thalmann 1983), uses a complete programming language enhanced for animation. Support is provided to model shapes and to move objects. A script, or program, is then written to generate the animation. Facilities can be provided to automatically iterate through time or control and coordinate multiple parallel processes. The second major class of system, exemplified by GRASS (Defanti 1973), BBOP (Stern 1983a, b; Sturman 1984) and GRAMPS (O'Donnell and Olson 1981), is interactive and picture driven. The model is positioned by the animator in real time and positions are stored in a database at key frames and interpolated to form the animation.

Both types of systems have advantages and disadvantages. The language approach is best for algorithmic movement or when the movement is to simulate a physical process, whereas the interactive approach tends to achieve more natural, personalized motions. An advantage of using the programmed language approach is that the animator is required to codify his algorithms, and therefore, as the system is used, new capabilities and tools are added. Of course, learning to use such tools requires animators with programming experience. The language-driven approach tends to be flexible because many different types of changes can, in principle, be controlled. High-level control strategies such as goal directed or functionally defined motions are likely to be language-based. However, the ultimate criterion is the quality of the final images, and since interactive systems provide immediate visual feedback, they encourage choices based on aesthetic values rather than implementation considerations.

We have been trying to combine the two approaches. Scenes are described with a geometric modeling language. The description includes the overall structure of the characters and the environment: the geometry of the different surfaces, the transformations which connect the different parts, and the constraints which allow the parts to be coordinated. A set of parameters is also declared; the final position, shape and characteristics of the model is a function of this set of numbers. An interactive program interprets this language and also provides an environment in which the animator can control the model. Users can tailor their interactive environment by designing their own screen menus, function keys, and command abbreviations. Most importantly, users can continuously modify different parameters by connecting their values to numeric input devices. The model is displayed on a real time vector display and its position changes continuously. The interpreter also controls a generalized parameter database. The database saves useful parameter sets, and also provides tools so that these parameters sets can be manipulated and recombined. Animation is generated by putting parameter sets in keyframes and interpolating the inbetweens.

Parametric models

The modeling language example in Fig. 1 describes a tube capped with two spheres. The length of the tube is "a" and its radius is "r". Figures 2 and 3 show the displayed model.

```
{% tube
 parameter scalar a=2, r=1  % initial values set to 2 and 1
 {% center
  rot -90 z         % rotate -90 degrees about the z axis
  scale r, 2*a, r   % scale in x, y, and z
  cylinder          % vertical cylinder of length 1, radius 1
 }
 {% left
  move -a, 0, 0     % move in x, y, and z
  scale r           % scale x, y, and z by r
  sphere            % sphere of radius 1
 }
 {% right
  move a, 0, 0
  scale r
  sphere
 }
}
```

Fig. 1. Model description of a capped tube

The language is block-structured, similer to Pascal or Algol. Each block is delimited by braces and can be given a name following the opening brace. *Tube*, *center*, *left*, and *right* are the block names in Fig. 1. Variables are local to the block in which they are declared and conform to Pascal-like scoping rules (that is, variables declared in an outer block are accessible to an inner block if not redeclared in the inner block, and variables declared in an inner block are inaccessible to an outer block). Primitive shapes are created by naming the generic type. Typical primitives are quadrics and polygons. Expressions can be used to obtain scalar, vector, or matrix values, and can be constructed from a wide set of logical and mathematical operations and functions. Transformation matrices are formed by the commands *move*, *rot*, and *scale*. Statements whose resulting value is a matrix cause that matrix to be concatenated onto the current transformation matrix. When a block ends, the current transformation matrix is popped off the stack.

A special class of variables are designated as parameters which can be used to modify the final shape of the model. Typical uses of parameters are to define joint rotations. For example:

```
{% Euler-joint
 parameter scalar phi, theta, psi

 rot phi x
 rot theta y
 rot psi z
 ...
}
```

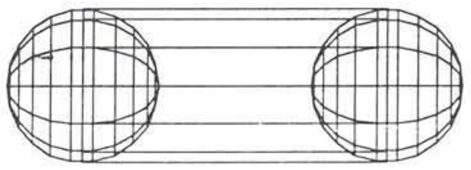


Fig. 2. Capped tube with a = 2, r = 1

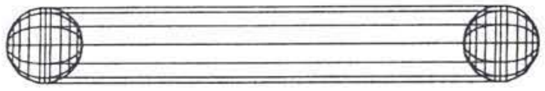


Fig. 3. Capped tube with a = 3, r = 0.5

parameterizes a joint by the three Euler angles which can be used to define an arbitrary rotation in three dimensions. Parameters also may participate in constraint expressions.

```
{
 parameter scalar phi

 rot limit (0, phi, 45) x
 ...
}
```

shows how the rotation about the x-axis is forced to lie between 0 and 45 degrees. Coordinating different parts of the model can be achieved by having them depend on a single parameter. To swing two arms in opposite directions, we can parameterize a model as

```
{% body
 parameter scalar wx
 {% left-arm
  rot wx x
  ...
 }
 {% right-arm
  rot -wx x
  ...
 }
}
```

Objects, as well as transformations, can be parameterized. A simple script interpolates between a triangle whose vertices are defined by the three vectors (a1,a2,a3) and another defined by the three vectors (b1,b2,b3) (the function *lerp* linearly interpolates between its second and third arguments based on a percentage expressed in the first argument):

```
{
 parameter scaler t
 vector a1,a2,a3
 vector b1,b2,b3

 triangle lerp(t,a1,b1), lerp(t,a2,b2), lerp(t,a3,b3)
}
```

The interpreter reads a model and then compiles a program that displays the model on a real time vector display (Evans and Sutherland Multi-Picture System). When the interpreter is running, any variable can be changed by simply assigning a new value to it. This causes an incremental re-execution of all the statements that depend on that parameter. To achieve fast updates, we build a dependency graph which lists the variables depending on each line and the lines depending on each variable. Therefore, some conventions must be followed when designing the model. For example, statements that produce cyclic dependencies are not allowed.
The interpreter accepts and immediately executes modeling language statements, as well as commands, to the interactive system. These commands can be typed in or read from a file. One group of commands is used to move to different blocks in the model in a manner analogous to traversing a transformation tree. At each particular block, only those variables in its scope are considered active and can be changed. Other commands are used to set up different interactive modes and to manipulate the database.

Interactive control

One of the main goals of this system is to make interaction as flexible and extensible as possible. We felt that the best approach would be to design an interpreter with a comprehensive set of commands, and provide tools with which the user can structure those commands into a personalized interface. The user dynamically loads this interface into the system. There are three primary tools for setting up interaction modes: a system of relating input devices to parameters, a menuing facility, and a method for programming keyboard and function keys.
To control parameters with input devices, we model inputs as varibles whose numerical values are functions of physical devices. Physical devices include a data tablet, a set of eight dials, and a three axis joystick (Fig. 4). Each of the x, y, and pen status (z) values of the tablet are variables, as well as mouse-x, mouse-y, and mouse-z values which correspond to the rate of change of tablet x, y, and z values. In addition we have a set of number wheel (Thornton 1979) variables connected to the x and y movement of the pen. From the joystick we get x, y, and z deflection values. From the dials we get absolute values as well as rates of change. There also is a logical "ticking" device which can be used to increment (or decrement) a value at regular intervals. Active input variables are monitored continuously. As they change, statements that de-

pend on them are re-executed and the model is modified on the vector display device. Statements relating parameters and input devices can be entered on the keyboard or read in from a file. To demonstrate this we modify the model description in Fig. 1 as follows:

```
{% tube
 parameter scalar a=2, r=1
 % initial values set to 2 and 1
 parameter scalar dx=0, dy=0, dz=0, spin=0
 move dx, dy, dz
 rot spin y        % rotation about y axis
 {% center
  ...
 }
 ...
}
```

Now we can control the movement of the tube with the following assignment statements:

```
dx  +=  wheelx
dz  +=  wheely
spin  +=  10*joyz
```

The left side of the assignment statement is a parameter declared in the model. The right side of the equation is an input variable. The right side of the statement may contain any expression in the modeling language. The three statements above increment the x-axis displacement of the tube by the value of the tablet x-axis number wheel, the *z*-axis displacement of the tube by the value of the tablet y-axis number wheel, and the spin (or rotation) of the tube by the value of the z-axis of the joystick. The statements are re-executed continually while the input devices are being used, so if we deflect the joystick on its z-axis, the tube spins in the direction of the deflection. If we want to constrain the displacement of the tube to lie along its longitudinal axis only, we can establish the following input mode:

```
dx  +=  5*wheelx*cos (spin)
dz  +=  5*wheelx*sin (spin)
spin  +=  10*joyz
```

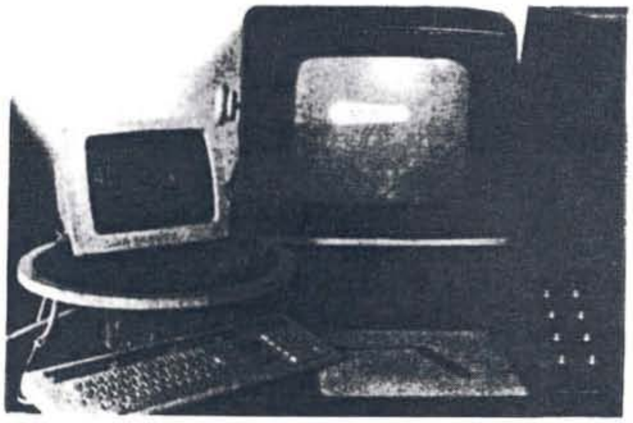


Fig. 4. Workstation showing terminal, keyboard, Evans and Sutherland Multi-Picture System, joystick, tablet and dials

At each block, any of the currently active variables can be related to input devices. This, for example, allows us to maintain control of the camera movement variables regardless of which block we are at in the transformation tree. Different sets of inputs can be specified for each block of the model. When the command interpreter enters a particular block of the model, it reads in a saved set of input assignments for that block. Accordingly, each block can have an input control with characteristics best suited for it: such as using the tablet to control motion of the whole body, and the joystick for an arm. In addition, the user can read in any set of input assignments allowing alternative input modes for the same parameter sets.

The menuing system allows users to create their own system of menus to interface with the command interpreter. A menu is a text file of items that can be displayed on the screen and selected using a tablet. The item picked is then input to the command interpreter. Commands can also be typed in. There is a command to switch to another menu so the user can organize a network or hierarchy of menus. Users may use the default menus, or create and organize their own menus on a per-model, or per-user basis. The command interpreter itself generates a set of menus that can be used to move to each block of the model, and to pick parameters of the model.

Finally users can "alias" keys and key-sequences. The alias facility allows users to assign commands to one key or sequence of keys. For instance, a frequently used command that is particularly long can have an alias that is either an abbreviation, or a single key on the keyboard. The command interpreter responds to the abbreviation or key exactly as if the full command had been given. Keys and abbreviations need not represent full commands or lines. Partial commands and strings can also be aliased. In addition sets of aliases can be saved and read in automatically.

Manipulating parameter sets

In addition to controlling the values of parameters by means of interactive input devices, there is a system for storing, recalling and manipulating sets of parameters. The hierarchical structure of the parameter database resembles the structure of the model defined in the modeling language. Each model block is represented as an association list whose value is the set of its parameters and inner blocks. Parameters are saved as name-value pairs. For example, the data for the lower body of a character (Fig. 5) might be represented as:

```
(lower-body
 (wy{0.0})
 (l-leg
  (wx{0.0})
  (wz{0.0})
  (lower-leg
   (wx{0.0})
   (foot
    (wx{0.0})
    (wy{0.0})
    (wz{0.0})
    (toes
     (curl{0.0})))))
 (r-leg
  ...))
```

(In this hypothetical model the wx's, wy's and wz's represent rotations about the x, y, and z axis.) In the printed representation of the database, parentheses enclose association lists, and braces enclose numerical data. Names following left parentheses correspond to names given to blocks and parameters in the model. Commands exist which read and write parameter sets from the database. For example, it is possible to write all the parameters that are within the current block and sub-blocks, or write all variables within the scope of the current block which match a regular expression. Besides these transactions, the data base can be stored in a text file and directly edited.

The current values of all the parameters are stored in a *working database*. By repositioning the model (updating the *working database*) and then copying all or a portion of the *working database* to a *states database*, we can develop a library of positions. When any one of those positions is re-read into the *working database*, the command interpreter executes the modeling statements depending on the changed parameters and updates the display. In this example, the default parameters for a lower body shown in Fig. 5 are stored under the name *resting*.

```
(resting
 (lower-body
  ...))
```

A subset of parameters which bend the knee can be stored as:

```
(bent-knee
 (lower-leg
  (wx{-45.0})
  (foot
   (wx{20.0})
   (wy{0.0})
   (wz{0.0}))))
```

The database allows parameter sets to contain pointers to other parameter sets. These are repre-

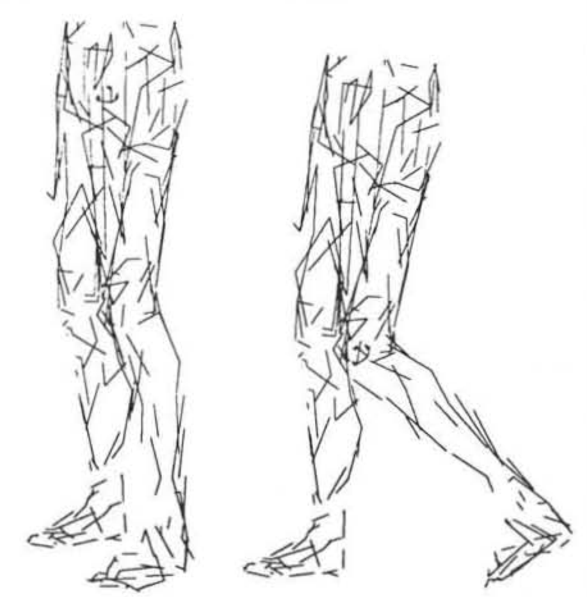


Fig. 5. Lower body in the resting or default position

Fig. 6. Lower body in the resting position with its knee bent 45 degrees

sented in the printed database as a name followed by a pair of empty braces. For example, we combine the two previous parameter sets into *position-1*:

```
(position-1
 (lower-body
  (resting { })
  (l-leg
   (bent-knee { }))))
```

This defines *position-1* as the lower body resting but with a bent knee (Fig. 6). When combining parameter sets, multiple values may appear for each parameter with the last values taking precedence. This is an important feature since it allows us to refine the position of the model by adding specific sets of parameters to general or default parameter sets.
Animation is generated from a special database which contains entries at keyframes. The knee bend described above can be animated by inserting two key frames in the animation database.

```
(animation
 (frame-1
  (body
   (resting { })))
 (frame-20
  (position-1 { })))
```

When reading from this database, parameter values between keyframes are interpolated automatically. Different interpolating functions, such as linear, cubic spline, and cosine, can be specified for individual parameters and keyframe intervals. It's important to note that parameters can be idependently key-framed.

Discussion

For five years the majority of NYIT's animation was done using the *BBOP* interactive keyframe system. Animators found it simple to use – each joint has nine degrees of freedom (3 rotation, 3 translation, and 3 scale) controlled by a 3-axis joystick. Some animations were programmed using *C* and then brought over to *BBOP* for fine tuning. We've found it faster and easier to use the interactive approach; however certain motions, like those of sophisticated robot models using Newtonian mechanics, multi-legged walk cycles, and terrain following, could be achieved with greater realism with a slower programmed approach (Lundin 1984). Both have served us well. *EM* was developed as an enhancement to *BBOP* to bring a form of programmed control to the keyframe approach.
We have used the system to control a range of models from articulated robots to parameterized face models (Parke 1982). Given the flexibility of the modeling language, complicated parametric models can be described and intricately animated. This was more difficult with *BBOP* where the effects of parameterization had to be simulated by hand.
One of the most successful features of the system is the ease with which input devices can be used to set the values of parameters combined with the power of real-time update of parametric models. Because of the intensity of the interaction between animator and computer, the design of control modes is as important as the design of the geometry of the model. For this reason the input system was designed to be programmable and reconfigurable. A group of parameters can be controlled by a single device, or a single parameter can be a function of several devices. Connecting functions of input devices to parameters also permits the construction of constrained input modes. We can create input modes where a robot rolls in the direction it is heading, or a camera rotates about either a local or a global axis. This flexibility can also be used to group the control of related parameters or establish alternative control modes for the same set of parameters. When controlling a parametric face model, there can be one input mode to control smiling and another for frowning; these two modes are likely to share parameters. Finally, the flexibility allows the input environment to be customized for the animator's comfort and convenience. Control can be transferred to those devices which feel most natural. Typically the camera motions are assigned to the joysticks, but when the joysticks are being used to control an articulating joint, camera control can be maintained by switching it to the dials.
Defining animations by time-varying parameters has its limitations. The position and shape of the model is a function of an instant in time. Currently there are no methods for accessing previous or subsequent values of a parameter, or its rate of change. This prevents us from incorporating dynamics and hysteresis in the model definitions or control. An-

other limitation of our system is that parameters control the geometry through simple statements and expressions; *for-loops* and cyclic dependencies are not allowed. For implementation reasons the display routines are forced to have a fixed size. As a result, it is not possible to change the arrangement of the transformation tree or topology of the model through time. Such changes are desirable in complicated animations.

Recently, there has been a great deal of interest in creating motion control systems that work at the functional level or are goal directed (Korein 1982; Zeltzer 1983). Although we did not originally design the system for these reasons, it has some features of these higher-level systems. Because we can enforce constraints and form dependencies, it is possible to design coordinated movements into the model. Parameters which have functional meanings can be created. For example, a parameter could control a walking cycle or leg-lift. This system could also be used to support a higher-level goal directed animation system. Such a system would interact with a large symbolic position database. Positions in the database could correspond to the achievement of subgoals, such as having the foot raised. With our database the current state of the system can be described in terms of these sub-states. A production system with a set of rules establishing conditions and the effects of transitions between these states in conjunction with a planning program could be used to automatically generate the keyframes in the animation.

Acknowledgements. This system is based on the program BBOP written by Garland Stern. Without this program to build upon this work could not have been done. The geometry language was developed by Jim Blinn and Tom Duff. The MPS subroutine library was written by Jim Clark and Garland Stern. The major routines used to display refresh buffers in real time were written by Thad Beier. The interpolation routines were based on Carl DeBoor's spline package. Robert Thornton stressed the importance of reconfigurable inputs and invented number wheels.

References

Defanti TA (1973) The Graphics Symbiosis System An Interactive Minicomputer Graphics Language Designed for Habitability and Extensibility. Ph. D. Dissertation, Ohio State University

Hackathorn R (1977) Anima II: a 3-D color animation system. Comput Graph (SIGGRAPH '77 Proceedings) 11:54–64

Hackathorn R, Parent R, Marshall B, Howard M (1981) An interactive microcomputer bases 3-D animation system. Proceedings of Conference of the Canadian Society for Man-Machine Interaction, pp 181–191

Korein JU, Badler NI (1982) Techniques for generating the goal-directed motion or articulated structures. IEEE Comput Graph Appl 2:71–81

Lundin D (1984) Motion simulation. Nicograph '84 Proceedings Tokyo, Japan

Magnenat-Thalmann N, Thalmann D (1983) The Use of High-Level 3-D Graphical Types in the Mira Animation System. IEEE Comput Graph Appl 3 (9):9–16

O'Donnell TJ, Olson AJ (1981) GRAMPS – A graphical language interpreter for real-time, interactive, three-dimensional picture editing and animation. Comput Graph (SIGGRAPH '81 Proceedings) 15:133–142

Parke FI (1982) Parameterized models for facial animation. IEEE Comput Graph Appl 2:61–68

Reynolds CW (1982) Computer animation with scripts and actors. Comput Graph (SIGGRAPH '82 Proceedings) 16:289–296

Smith AR, Cook R, Carpenter L, Porter T, Salesin D (1983) Road to Point Reyes. Title Page Credit. Comput Graph 17

Stern G (1983a) Bbop – a system for 3D keyframe figure animation. SIGGRAPH '83, Course 7, Introduction to Computer Animation, pp 240–243

Stern G (1983b) Bbop – a program for 3-dimensional animation. Nicograph '83 Proc Tokyo, Japan, pp 403–404

Sturman DS (1984) Interactive keyframe animation of 3-D articulated models. Proc Graph Interface '84 Ottowa, Canada, pp 35–40

Thornton RW (1979) The number wheel: a tablet based valuator for interactive three-dimensional positioning. Comput Graph (SIGGRAPH '79 Proceedings) 13:102–107

Tucker JB (1984) Computer graphics achieves new realism. High Technol 4 (6):40–53

Zeltzer D (1983) Knowledge-based animation. Proc ACM SIGGRAPH/SIGART Interdisciplinary Workshop, Motion: Representation and Perception. Toronto, Canada, pp 187–192

MOTION SIMULATION

Richard V Lundin

New York Institute of Technology
Old Westbury, New York

ABSTRACT

The BBOP animation system developed at the New York Institute of Technology was designed to animate hierarchically articulated 3-d models by interpolating key frame poses established during interactive sessions with an Evans and Sutherland Picture System. This system was evaluated in a research project in which 3-d robot characters were animated in a simulated environment with the goal of achieving a realistic rather than a cartoonish look. In the course of the project various techniques evolved for performing tasks - not achievable within the framework of BBOP - as a post-process to the BBOP animation. These tasks include the creation of motion for wheeled, tracked and multi-legged robot vehicles over uneven terrain, the simulation of dynamics effects on robot motion, and the animation of models with flexible parts. This paper describes those techniques.

MOTION SIMULATION

Richard V Lundin

1. INTRODUCTION

A research animation project at video resolution has been under way at NYIT to evaluate the 3-d animation system BBOP [1,2]. . Using this system, an animator poses a tree-structured articulated model by means of interactive devices to create key frames. The key frames are then interpolated by cubic spline techniques to produce the remainder of the frames.

The scenario for the animation is rather mundane in nature, a gang of construction robots assemble a communications satellite antenna, but it explores new territory by requiring 3-d models to manipulate objects and operate in a fairly realistic manner in a simulated outdoor environment. The robots move about by all manner of locomotion: some are wheeled vehicles, some tracked vehicles and some are multi-legged vehicles resembling ants. To animate these vehicles such that the wheels rotate correctly, the wheels and feet make contact with the ground, and the vehicles jerk and bounce in response to movement over the terrain would be a challenging task indeed using only the BBOP system. This class of motion cannot be adequately described by an intepolating procedure since the motion parameters must be determined on a frame-by-frame basis according to specific equations or procedures, more appropriately performed by *simulation algorithms.*

The BBOP data base consists of the tree-structure for the model and the values of parameters to each of the transformations defining the position and orientation of each articulated part of the model for each frame of the sequence. Routines are available for reading the data base and performing operations such as determining the matrix describing the transformation from one joint in the tree to any other joint. By accessing the data base any simulation can be performed as a post-process to the BBOP animation.

The procedure for implementing a simulation algorithm is to first animate the model as much as possible with the BBOP system. For example, a vehicular model is moved over a path, but the wheel motion is not performed. The simulation program is then invoked which reads the BBOP data base, performs the simulation, and writes the modified transformation parameters back to the data base. The effect of the simulation on the animation can then be viewed on

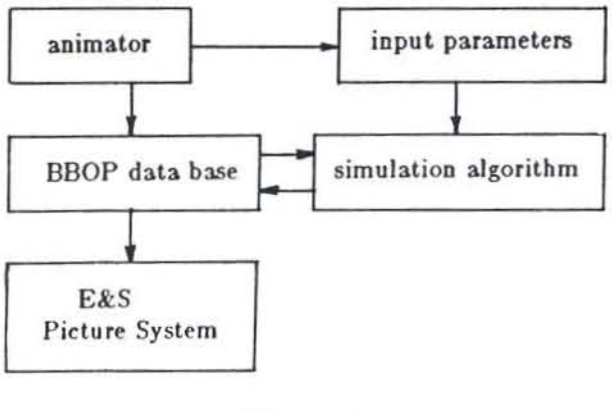


Figure 1

the Evans and Sutherland Picture System, and further modifications made if required by either changing input parameters to the simulation algorithm or by changing the motion data base with BBOP. In fact simulation algorithms can be tuned to achieve exaggerated motion if a cartoonish style is desired. The implementation of a post-process simulation algorithm is illustrated in the block diagram of figure 1.

Various simulation algorithms developed during this project will now be described.

2. GROUND SURFACE CONTACT

Common to all types of earth-bound (although any planet will do) models is the requirement of maintaining contact between the model and the ground surface. The ground terrain model usually consists of a texture-mapped polygonal mesh on which are placed various props that establish the set. A vector model of the terrain is created to be used on the Picture System. The moving model or character is animated over the terrain with the BBOP system to establish the path and speed of the model. The terrain vector model is usually too coarse to allow the animator to establish contact of the wheels or feet of the model with the terrain or to orient the model with respect to its path.

Some scheme must be conjured that establishes ground contact as a post-process to the animation. One method is to take advantage of the polygonal aspect of the terrain and calculate the equation of the plane

$$Ax + By + Cz + D = 0$$

for each triangular polygon of the mesh. The intersection point of a vertical vector or plumb line from the axis of a wheel or from an ankle to the plane below can then be calculated from

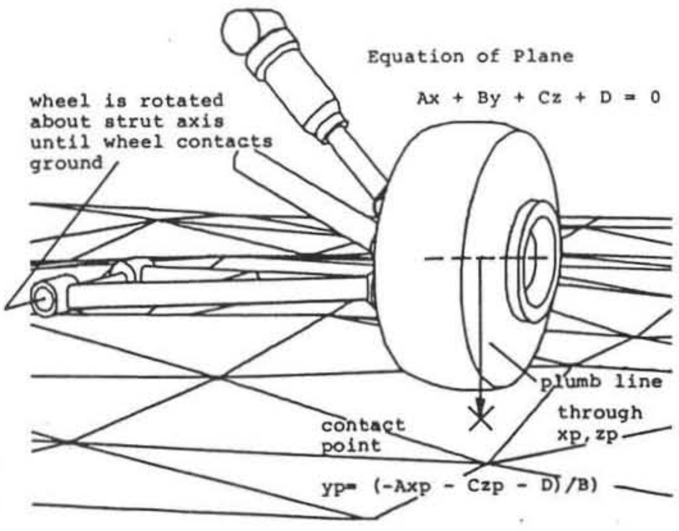

Equation of Plane
Ax + By + Cz + D = 0
wheel is rotated about strut axis until wheel contacts ground
plumb line through xp, zp
contact point
yp= (-Axp - Czp - D)/B)

Figure 2

$$y_p = (-Ax_p - Cz_p - D)/B$$

where x_p, z_p is a point through which the plumb line passes and y_p is the height of the terrain (figure 2). The vector equation of the normal

$$\vec{N} = A\hat{i} + B\hat{j} + C\hat{k}$$

to the plane can also be used to re-orient a foot so that it is flat on the ground.

If z-buffer-based rendering programs are available, this method can be adapted to use the z-buffer instead of the polygonal mesh. The ground terrain model is rendered from above and the resulting z-buffer is used to divine the topology of the surface. This method allows the surface to be modeled by any means and can be littered with all sorts of paraphernalia - which can be run over or trod upon. The plumb line is transformed to a pixel location in this *z-space*, and the z-value at this coordinate, when transformed back to *world space*, is the height of the terrain. The normal to the ground at this point can be found from the heights at adjacent pixel locations, using conventional cross-product techniques.

The contacting surface - the bottom of a wheel or sole of a foot - is moved to the contact point by adjusting the appropriate transformation parameters according to the geometry of the model. If the terrain model represents a deformable surface like dirt or sand, the contact point can also be lowered, such that a whẽel or foot appears to sink into the ground thanks to z-buffer technology.

Wheel tread marks or footprints can be registered on the ground terrain in the rendering process by implementing the bump map option of the polygon rendering program with a map consisting of wheel paths or footprints created by rendering the model from below.

3. WHEELED AND TRACKED VEHICLES

The path of a wheeled vehicle such as "Driller" (figure 8) over an uneven terrain can be approximated by a 2-d path in x and z derived from the initial BBOP animation. Furthermore, the path of the centerline of the vehicle can be approximated by a series of arcs, each arc calculated from three points in the 2-d path [3]. The amount of wheel rotation per frame can then be prescribed by dividing the arc distance traveled by each wheel during the frame by the radius of the wheel as demonstrated in figure 3 in which "Driller" is shown from above. The steering angle of the vehicle's steering wheels can also be easily calculated from the geometry of this

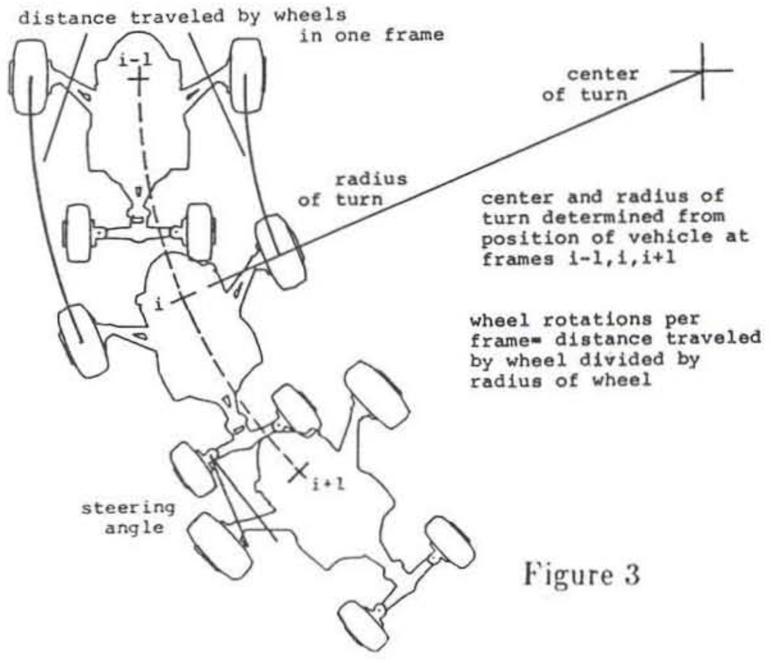
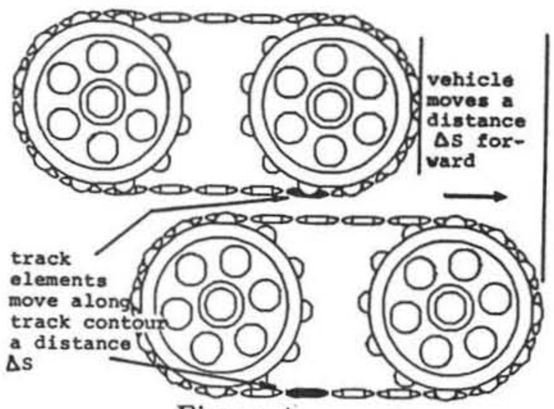

distance traveled by wheels in one frame
i-1
center of turn
radius of turn
center and radius of turn determined from position of vehicle at frames i-1,i,i+1
i
wheel rotations per frame= distance traveled by wheel divided by radius of wheel
i+1
steering angle

Figure 3

configuration. The vehicle itself is reoriented at each frame to point to its position at the next frame.

The animation of tank-like tracks is derived from the arc distance traveled each frame by the centerline of the track. Each track element is displaced by this distance along the track contour formed by the wheel system that supports the track as illustrated in figure 4.


vehicle moves a distance ΔS forward
track elements move along track contour a distance Δs
Figure 4

4. WALKERS

A general walking program has been developed to animate any model with legs consisting of two linked members such as "Humpty" shown in figure 5. Input to the program consists of parameters such as stride length, heel strike angle, toe-off angle, and timings for step events to define the walk characteristics of the model.

The animator provides the motion for the torso of the model on the BBOP system. The walk program is then invoked to create the motion of the legs from purely geometrical considerations such that the supporting feet contact the ground and swinging legs clear the ground.

The description of the stance phase of a leg is based on a solution to equations describing the distance from the hip joint to the ankle joint. The text file describing a subtree consisting of one leg of "Humpty" is:

```
{
move 43,3.2,46
rot 30 x
rot 26 y                [M]
rot 26 z
scale 0.65,0.65,0.65
"torso"
        {
        move 1,0,0
        rot x
        rot y
        rot z
        "thigh"
                {
                move 0,-1.5,0
                rot x
                "calf"
                        {
                        move 0,-1.5,0
                        rot x
                        rot y
```

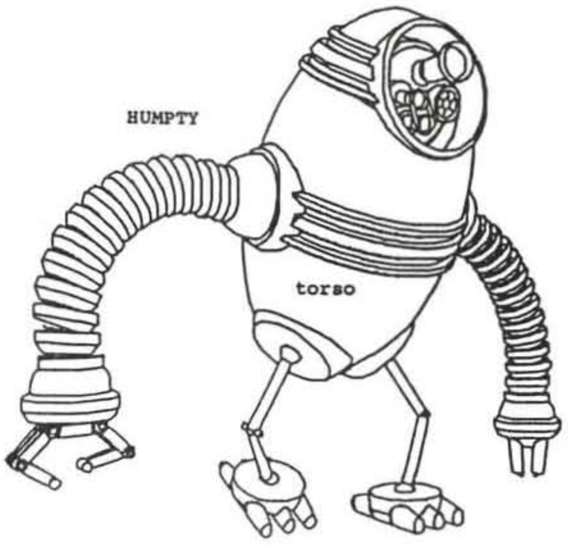


Figure 5

```
                            rot z
                            "foot"
                            }
                      }
                }
    }
    {
    "terrain"
    }
```

To establish contact between the bottom of the foot and the ground terrain, the rotation angles for the hip, knee, and ankle joints must be determined for each frame (to fill in the missing arguments of the rotation transformations in the text file above). The hip joint and the ankle joint are considered to be ball joints and the knee joint is considered to be a pin joint. The position of the hip joint P_h with respect to the *torso* coordinate system at each frame is calculated from the BBOP data base for the animation of the *torso*. The ground contact point of the foot P_f with respect to the *world* coordinate system is first determined from one of the two methods previously described. The foot is aligned along the vector normal to the ground at this point. The coordinate of the ankle joint P_a with respect to the *world* coordinate system can then be determined from the equation of the line in the direction of the normal vector as shown in figure 6. The coordinate of the ankle joint P_a' with respect to the *torso* coordinate system is determined by transforming the ankle coordinates P_a with the inverse of the matrix $[N]$, formed by concatenating all transformations from the *world* coordinate system to the *torso*.

$$[P_a'] = [P_a][N]^{-1}$$

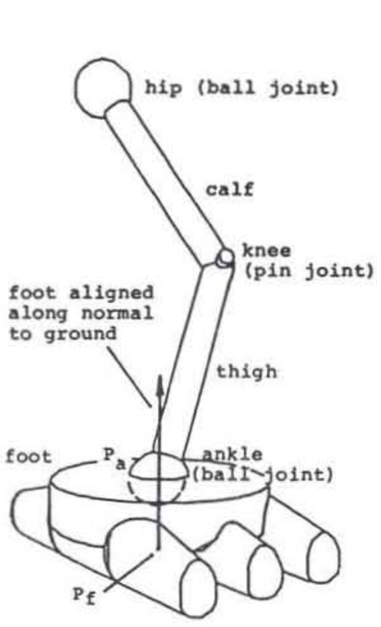


Figure 6

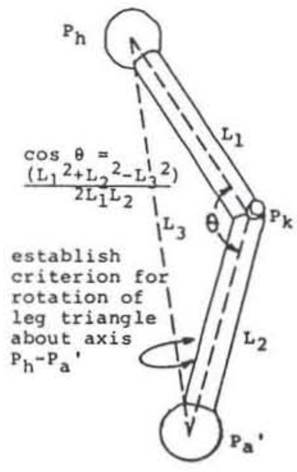

P_h
L_1
$\cos\theta = \frac{(L_1^2+L_2^2-L_3^2)}{2L_1L_2}$
L_3
θ
P_k
establish criterion for rotation of leg triangle about axis P_h-P_a'
L_2
P_a'

Figure 7

The transformed coordinate Pa' is one vertex of a triangle through the hip joint, knee joint, and ankle joint. Recourse to the law of cosines determines the inscribed angles of the triangle shown in figure 7. Since the triangle can be rotated arbitrarily about the axis defined by the vector from the hip joint to the ankle joint (that is, the system is underconstrained), some criterion must be used to determine the orientation of the triangle about this axis. One criterion is to retain the relationship between the triangle formed by the legs and the direction of motion of the model that was established in the original BBOP animation. Once the orientation of the triangle is established, the coordinates of the knee P_k can be determined and subsequently the required angles at the hip, knee, and foot.

The determination of the rotation angles for the swing phase of a leg is based on an interpolation between the position of the leg in its last stance phase and the position in its next stance phase.

5. DYNAMIC SIMULATION

Since "Driller" (figure 8) is mounted on a suspension system consisting of struts and shock absorbers, it should bounce up and down as it moves over uneven terrain, roll to one side as it turns, and pitch back and forth due to acceleration in the direction of its path. A simple dynamics model based on damped mass-spring systems [4] was formulated to provide these responses.

The forces due to acceleration on the vehicle were approximated by the equations of motion for a rigid body rotating in a circular arc about a fixed axis [5]:

$$F_r = mr\,\omega^2$$

$$F_t = mr\,\alpha$$

in which:
- F_r = force radial to arc
- F_t = force tangential to arc
- m = mass of body
- r = radius of arc
- ω = angular velocity of body
- α = angular acceleration of body

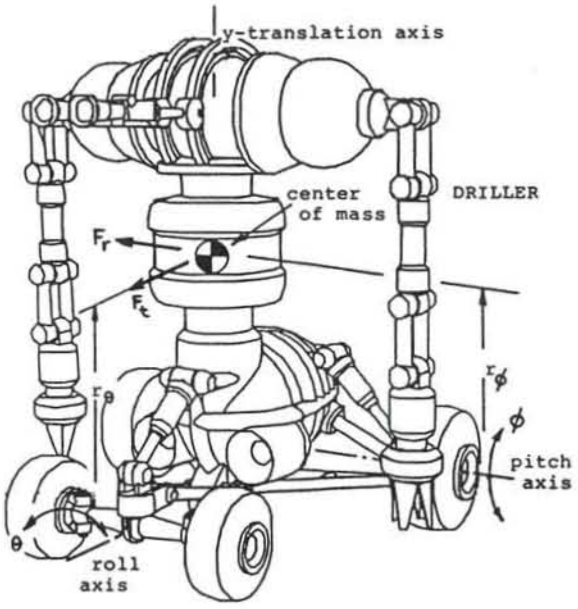


Figure 8

The values of r, ω, and α can be calculated for a path that has been approximated by a series of arcs. The dynamic response of the vehicle due to these forces depends on the geometry of the model and are determined by balancing force and torque equations. All of the elastic forces (eg., forces from springs) and all of the damping forces applied to the vehicle by its suspension system are lumped in single terms in the force and moment balances for this simplification.

Balancing forces in the y-direction gives the following equation:

$$m\frac{d^2y}{dt^2} = F_y - Ke_y\,y - Kd_y\,\frac{dy}{dt}$$

in which:
- m = mass of vehicle
- y = vertical position of center of mass
- F_y = applied vertical force proportional to vertical acceleration due to motion over uneven terrain
- Ke_y = elastic coefficient (spring constant) for vertical translation
- Kd_y = viscous damping coefficient for vertical translation

$t =$ time

Balancing torques about the pitch or roll axis gives the following equation:

$$I_\phi \frac{d^2\phi}{dt^2} = F_\phi r_\phi - Ke_\phi \phi - Kd_\phi \frac{d\phi}{dt}$$

in which:

$I_\phi =$ moment of inertia of vehicle about pitch or roll axis
$\phi =$ angle of rotation of vehicle about pitch or roll axis
$F_\phi =$ torque force (F_r for roll and F_t for pitch)
$r_\phi =$ radius from axis where force F_ϕ is applied
$Ke_\phi =$ elastic coefficient for rotation about pitch or roll axis
$Kd_\phi =$ viscous damping coefficient for rotation about pitch or roll axis

It should be noted from these equations that if the vehicle is translated or rotated from its rest configuration by the forces due to motion, forces of opposite sign due to elasticity and damping will tend to restore the vehicle to its equilibrium state. By converting the differential equations into difference equations, they can be solved to provide the translational and angular position of the vehicle for each frame of the animated sequence:

$$m\left[\frac{(y_i - y_{i-1})}{\Delta t} - \frac{(y_{i-1} - y_{i-2})}{\Delta t}\right] / \Delta t = Fy_i - Ke_y\, y_i - Kd_y \frac{(y_i - y_{i-1})}{\Delta t}$$

$$I_\phi\left[\frac{(\phi_i - \phi_{i-1})}{\Delta t} - \frac{(\phi_{i-1} - \phi_{i-2})}{\Delta t}\right] / \Delta t = Ft_i\, r_\phi - Ke_\phi \phi_i - Kd_\phi \frac{(\phi_i - \phi_{i-1})}{\Delta t}$$

or

$$y_i = \left[Fy_i + y_{i-1}\left(\frac{2m}{\Delta t^2} + \frac{Kd_y}{\Delta t}\right) - \frac{m}{\Delta t^2} y_{i-2}\right] / \left(\frac{m}{\Delta t^2} + Ke_y + \frac{Kd_y}{\Delta t}\right)$$

$$\phi_i = \left[Ft_i\, r_\phi + \phi_{i-1}\left(\frac{2I_\phi}{\Delta t^2} + \frac{Kd_\phi}{\Delta t}\right) - \frac{I_\phi}{\Delta t^2}\phi_{i-2}\right] / \left(\frac{I_\phi}{\Delta t^2} + Ke_\phi + \frac{Kd_\phi}{\Delta t}\right)$$

in which:

i= subscript for variables evaluated at frame i
i-1= subscript for variables evaluated at frame i-1
i-2= subscript for variables evaluated at frame i-2
$\Delta t =$ time step (reciprocal of frame time 1/30 or 1/24)

The evaluation of the constant coefficients in these equations - mass, moments of inertia, elastic and damping coefficients - is performed during interactive sessions using the Picture System and no actual correspondance with the actual physical properties of the suspension system is required. The only thing of consequence is the achievement of the desired dynamic response.

6. EFFECTS OF GRAVITY

Because computer-modeled objects are in a sense "cut loose" from gravity, it is necessary to re-establish the reins of gravity. The formation of sparks due to a welding operation, for example, required the implementation of the following ballistic equations to determine the trajectory of the sparks:

$$y = -\frac{1}{2}gt^2 + v_{y0} + y_0$$

$$x = v_{x0}t + x_0$$

$$z = v_{z0}t + z_0$$

in which:
- $x, y, z =$ position of object at time t
- $g =$ gravitational constant
- $x_0, y_0, z_0 =$ initial position of object
- $v_{x0}, v_{y0}, v_{z0} =$ initial velocity of object in x,y,z-direction

The initial velocity and direction of the sparks were determined by stochastic means. The z-buffer of the ground terrain was used to find the location of the impact points of the sparks and new trajectories were computed following a bounce.

The determination of the trajectory of a thrown projectile was another example in which the effects of gravity were necessary. The initial position and velocity of the projectile were determined from BBOP animation from the position of the throwing hand at the frame at which the projectile was released and the position at the preceding frame. The position of the projectile for succeeding frames was then determined from the ballistic equations.

7. FLEXIBLE JOINTS

Another example in which a post-process technique is required is the animation of flexible joints. The BBOP program was originally intended to animate a tree-structure of rigid parts, therefore, a gap can appear at the joint of two connecting rigid parts when they are rotated with respect to each other. For example, the knee joint between polygonal representations of the thigh and calf of a body model will reveal a gap when the knee is bent. What is required is a flexible surface that will deform according to the rotation angles of the joint in the BBOP animation. One solution is to treat the joints connecting rigid segments as control points for a spline defining the centerline of a flexible segment, which is populated along its length with elements making up the flexible joint. The elements can be rigid models as would be incorporated in the goose-neck joint of the robot arm in figure 9 or contours defining the coordinates of rows of a flexible polygonal mesh. A bezier spline was chosen for this application because it lies on, and is tangent to, the centerline of the rigid segments defining each end, which is the configuration needed for a flexible elbow or knee. The disadvantage of this approach is that the length of the flexible joint changes as it bends.

If it is necessary to constrain the length of the flexible joint and the tangential end conditions are not essential, then it is possible to relocate the control points by iterative means for each frame to maintain a constant spline length as illustrated in figure 10.

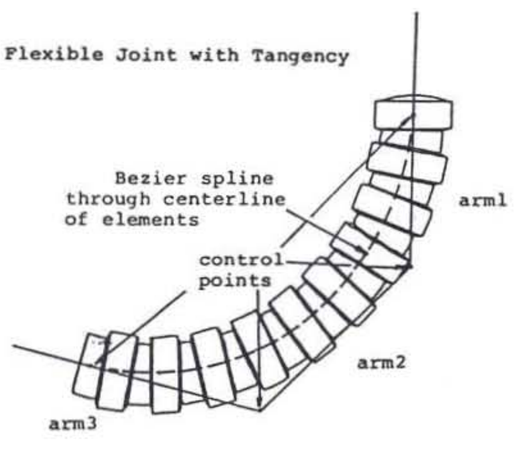

Flexible Joint with Tangency
Bezier spline through centerline of elements
control points
arm1
arm2
arm3

Figure 9

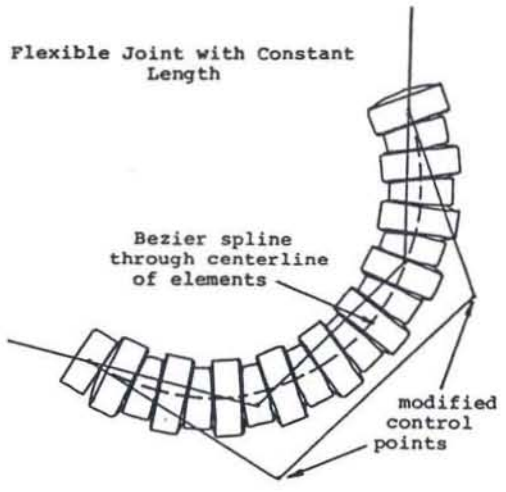

Flexible Joint with Constant Length
Bezier spline through centerline of elements
modified control points

Figure 10

8. CONCLUSION

A videotape sequence depicting a robot construction scene demonstrates that the application of simulation algorithms as a post-process to animation created with key-frame interpolating systems provides the framework for greatly enhancing the realism of animation. Moreover, simulation algorithms can be implemented for any imaginary world subject to any hypothetical laws of physics to achieve any desired effect.

REFERENCES

1. Stern, Garland, "Bbop - A Program for 3-Dimensional Animation", Proceedings of Nicograph '83 Conference, Tokyo.
2. Sturman, David, "Interactive Keyframe Animation of 3-D Articulated Models", Proceedings of Graphics '84, Ottawa, Canada, pp. 35-40.
3. Faux, I.D. and Pratt, M.J., "Computional Geometry for Design and Manufacture", Ellis and Horwood, Chichester, England, 1980, p.67.
4. Meriam, J.L., "Mechanics", John Wiley and Sons, New York, 1959, p.336.
5. loc. cit. ref 4, p.167.

SIGGRAPH '86 Tutorial Notes
Computer Animation - 3D Motion Specification and Control

Glenn Entis

Pacific Data Images
1111 Karlstad Drive
Sunnyvale, California 94089
(408) 745-6755

ABSTRACT

A **Script** for a computer generated animation is a sequence of instructions which define the animation. In its most general sense, the script is a program in a special purpose animation language.

This talk covers some of the general design principles and specific tasks of a script system. The PDI script system is used to demonstrate these principles in the video presentation which accompanies the lecture. This lecture is meant to be an introduction to the concept of scripted animation. It does not delve very deeply into any of the specific details of the animation process, but instead attempts to give a clear and visual overview of the script process.

In addition to presenting the flavor of a script system, this talk is also an exposition of the point of view that programming is a valuable animation skill. There are some animation techniques, situations, and disciplines which inherently require skills normally associated with programming. Non-programmatic and interactive tools are absolutely essential for fine animation as well, but discussion of those is beyond the scope of this paper.

The notes given here are companions to the live and video presentation.*

1. What are we talking about?

What is motion?

anything which changes during the course of an animation. Many interactive systems allow for the specification of flight paths, but animation can also include the animation of colors, material types, model parameters, viewing specifications, control variables, etc. Any element or value which contributes to the definition of a scene is animation fodder.

What is a computer animation?

Of course, any motion sequence generated through use of a computer. The important thing to keep in mind is that many interesting complex computer animations require more from the animator than the simple coordination of a few flight paths. Most commercial animations embody quite a few predictable elements, such as animating lights, special effects, coordinated events, etc. In addition to these elements, most animations have some surprises, "special features", or "wow, that's really hard" parts. In general, these hard parts tend to be those portions of the specification which elude easy generalization and thus simple tools.

*This paper originally appeared in the 1986 Course Notes and was accompanied by a lecture by the author. It is felt that its content is useful even without the lecture and so is reprinted here. -ed.

What is a script system?

The term **script** is used because 1) it sounds better than "special purpose animation language", and 2) its generality allows for the inclusion of support tools which augment the language, and/or tie it to interactive tools. A script-based system generally supplies high-level programming language features as well as special purpose features to support animation (see [Reynolds82] - reprinted for this tutorial - for an excellent discussion of scripts for computer animation). In general, we will define a script simply as a special purpose animation programming language. (see [Reynolds85] for a discussion of the different senses in which a scripting may require programming).

2. Why use a script system?

It is possible to make some very interesting computer animations without programming by using interactive tools found in a number of research and commercially available systems. We will lump these tools togther under the name of non-programming tools. These include motion editors, menu-based modellers, many keyframe systems, etc. [O'Donnell81, Gomez85]. Even in systems which rely on programming for some part of the specification, interactive tools are absolutely essential for getting the feel of a move, the right curve, or for quickly relating various elements.

If it is possible to animate without programming, why bother with programming based tools at all? There are several good reasons to provide serious support to a programatic control of animation:

1) Many animations in some way or another fall outside the design model embodied by currently available interactive tools. Even some tasks which can be accomplished from within a non-programming tool require a good deal of contortion and could more directly and easily be expressed in an animation language.

2) A programming approach to problem solving can be flexible, extensible, and efficient.

3) Many situations in computer animation are essentially procedural. Although some non-programming systems allow simple procedural control of an animation (such as making one control value a function of another), most non-trivial procedures require real control structures, handy named variables, shared subroutines - in short, a programming environment.

4) When a technique used for a single animation is found to be generally useful, it can be easily packaged for general use. An important point on this topic is the fine line which exists between "special purpose, non-general hacking", which is a bad thing, and "flexible, quickly customizable, well-focused solution" which is a good thing. Because several assumptions can be made about the animation environment, it is much easier and faster to develop an animation script than a functionally equivalent C program. Although many scripts are hacked (i.e. made quickly to solve a specific problem, without much forethought to organization or generality), they're hacked quickly and can be changed quickly. They also tend to be specific to a single animation and animator, which means they avoid the major software problem of hacky code masquerading as a public tool. When solutions embodied in scripts are seen to be generally useful, they are cleaned up and installed in a script library, or rewritten in a general purpose programming language and released and documented as a general tool. Although techniques learned on non-programmatic tools can be shared, the sharing is generally word-of-mouth rather than a common library.

The key to good animation system design is the integration of all the tools in the kit, so that interactive tools are available where appropriate, without preventing the animator from easily retreating to some underlying or organizing representation. Non-programmatic tools are well suited for many of the specific and predictable problems encountered in animation, but they often aren't adequate. If these tools can coexist with a programming, or script, environment, then the animator has the option of augmenting the non-programmatic tools with the specific control gained from a script system. This kind of "open-architecture" approach, where an underlying representation is shared by several different kinds of tools, doesn't constrain the animator to one particular model of the animation specification process.

3. Animation and Software Engineering

There are strong similarities between writing an animation script and writing a program in a conventional language, and between creating a complex animation and software engineering. The discipline of software engineering has a lot to teach animators in this area, and a good animation system should be able to exploit these lessons wherever possible.

An animator doesn't necessarily have to be a programmer to take advantage of these principles. However, while an animator may not have to program to make Donald's arm move, he may need the same talents as a good software engineer to make Huey, Dewey and Louie run around the room for 3 minutes. For all of its expression and flair, complex animation has always required huge amounts of patience, organization, precision and repeatability.

Some general software engineering principles which apply to animation are:

1) Sharing general techniques in public libraries, as mentioned above. In addition to the general desirablity of sharing, scripts allow work to be chunked as routines, which, like any chunking, provides good conceptual shorthand and a handy unit for documentation.

2) Management of large amounts of data. A recent PDI animation had 26 seconds of 60 fields each, hundreds of moving parts, and a total of 50 separate component layers which were composited together. All this for one of those "simple logo moves". Software engineering tools and techniques have evolved to help software developers create, track, organize, share, and back-up large amounts of data.

3) Maintaining complex organization throughout unpredictable changes. Complex systems are riddled with interdependecies. They are always built upon certain assumptions. These assumptions almost always change midway into the project. Experienced animators and software engineers share the ability to organize their projects in anticipation of the unexpected.

4) Clear, self-documenting organization. Software engineers begin by learning documentation skills. As they mature, they learn how to write clear code with descriptive variable and routine names. They nest their code so that it reflects its own logical structure. They organize code into functionally clear routines with minimal side-effects, and group these into functionally related libraries. All very wonderful, but hard in practice. These skills take time to develop, and rely on environments which allow them to be practiced in the first place. Protecting animators from some of the requirements of software engineering may also inadvertantly "protect" them from the opportunity to learn tools and techniques which are well-suited to higher levels of organization.

The argument presented in this section is that software engineering skills are **intrinsically** related to the animation process, and part of the potential application of computers is to provide tools which allow an animator to develop these skills.

4. Useful Features in a Script

Many of the useful features in a script apply to programming languages in general. Many of the good strategies in creating script-based animation apply to good software engineering in general.

What features are useful in a script? Here are several:

- allows fast prototyping of images and animation

- simplicity of design and use
- support of common graphics operations (primitive modelling, 3D transformations, viewing operations)
- high-level programming language functionality (control structures, named variables, routines, data typing, math expressions, etc.)
- support abstraction by allowing any functional operation to be applied to any graphic operation, object, or attribute
- some hooks to interactive tools

5. The PDI Script System

The PDI script system is patterned after the programming language C, and includes several features which support the specification of high quality animation for the broadcast market. Many of these features are covered in the accompanying video, as the PDI script is used to illustrate various aspects of the scripting process.

We also make heavy use of the UNIX† environment in the production of animation. Much of UNIX has evolved to simplifiy the process of software creation, and these same features are applicable to animation, particularly with regard to script creation and management.

6. The Animation Pipeline

Before we can proceed to discuss the deatils of animation and script systems, it is useful to look at the data path from original specification to display. The pipeline is a good model for this flow (see [Chuang83] - reprinted with course notes - for a description of this pipeline).

The PDI system has the standard entourage of world space and screen space polygon files, image manipulators, etc. This part of the discussion will describe how user specifications find their way to screen, how they can be modified (which includes animated) along the way, and how a script supports all this.

7. Script in Action

This section is a set of animated demonstrations of a script. The animation consists of a scene in motion, with one or more lines of the controlling script displayed at the bottom of the screen. As the line of script text changes, the scene is updated to reflect the change.

After each specific script feature is demonstrated in this manner, a piece of commercial animation will be shown which exploits that particular feature.

7.1. Primitive Generation

A script should be able to generate a variety of geometric primitives, including polygons, prisms, cylinders, cones, pyramids, spheres, torii, surfaces of revolution, and extrusions. Primitives created from a script can be a useful modelling components, since the script also provides the attribute and transformation commands needed to combine these primitives into meaningful models.

Also useful is the ability to animate primitive attributes over time. For example, the cross section of a cylinder describes a circle. If partial cylinders are supported (those whose cross-sections describe a circle with a wedge cut out of it), then the shape of the cylinder can be animated.

Simple geometric primitives can account for only a small portion of the models necessary for commercial

†UNIX is a Trademark of Bell Laboratories.

production. More complex operations (such as Boolean operations on primitives, see [Beier85]), models created from digitized input, and control at the polygon and vertex level should also be included in or be accessible from the script.

7.2. 3D transforms

The basic 3D transformations are **translate, rotate,** and **scale.** These are concatentated as they are invoked, so that the effects of several transformations may be applied to an object at once. An important feature for hierarchical motion is the ability to **push** and **pop** the current transformation onto and off of a transformation stack.

7.3. Object Transformations

More sophisticated transformations can make the script a powerful modelling tool. Non-linear transformations such as **twist** and **bend** have wide use and can applied to arbitrary input objects [Barr84]. Some commercially available systems have introduced **bevelling** as a basic modelling operation. **Shape interpolation** is another powerful tool, but in the most general case a good deal of specification is required for good behavior throughout the interpolation.

7.4. Viewing Operations

Viewing operations define the **camera, window,** and **viewport.** As with all other aspects of the system, these should be animatable. A camera can be defined by its position, **focal length** or **angle-of-view,** and orientation. An alternative to orientation is to provide a second position which the camera will **look at.**

The **window** and **viewport** define the visible area in world and screen space, respectively. The viewport commonly maps the entire display area, but can be shrunk to make low resolution tests.

7.5. Lighting

There are a wide variety of lighting options available to the animation system designer. The simplest is the **ambient light,** which illuminates surfaces independently of their orientation. **Infinite point sources** have a direction vector and color but no position or falloff. These are often adequate for modelling large scale lights such as the sun, but are not adequate for representing the effect of a light within a scene. **Local lights** do have position, falloff, and possibly direction, and more accurately simulate the local effects of a small light.

Whatever kinds of lights are available from the script, it is essential that all parameters controlling the lights be animatable. Sometimes a blatantly moving light is needed for a specific reason. However, it is more common that a carefully lighted animation requires subtly moving lights just to get everything to look good. In these cases, it may not be apparent to the viewer that the lights are moving, yet the overall result is a much better lighting throughout the piece.

7.6. Surface Attributes

The **surface properties** and **colors** determine how an object will respond to light. A good system should provide a means for fast guesses so that a prototype image can be quickly realized, but it must also support the grueling fine-tuning that is required of a polished production. As with all other attributes, these attributes must be animatable.

7.7. Motion Paths

Motion paths are curves in space. Objects, the camera, the camera tracking point, and lights may all follow these curves through the scene (see [Shelley83]). Since positioning and motion design are so heavily dependent upon the interaction of multiple objects within a scene, it is strongly recommended that motion design be done interactively wherever possible. The job of the script system is to easily accept as input the output of motion design programs, and to provide ways of further manipulating that output.

8. Conclusion

Script based animation systems provide the powerful model of programming to the animation process. Programming is precise, repeatable, provides powerful control structures, and can promote large scale organization skills exemplified by software engineering.

No single tool, or general model for tools, is in itself adequate to the task of creating high quality, complex computer animation. Much of the process is highly immediate, visual, and interactive in nature, and non-programmatic tools are desirable for these tasks. However, there are many animation problems which embody a complexity of logic or scale of organization which call for programming tools.

The approach advocated here is for a hybrid environment in which script systems - programming tools - are supported and used as a necessary component of the complete animator's tool kit.

9. References

In addition to the references cited here, a relatively new development is the commercial availablity of 3D animation systems. Although little information has been published about these systems, it is suggested that the serious student follow up with requests for product brochures, user manuals, sample video tapes, and live demonstrations.

1) Barr, Alan H. "Global and Local Deformations of Solid Primitives", *SIGGRAPH '84 Conference Proceedings*, **18** pp. 21-30 (July, 1984).

2) Beier, Thaddeus "Object to Object Clipping" SIGGRAPH '85 Tutorial on *State of the Art in Image Synthesis* (August, 1985).

3) Chuang, Richard and Glenn Entis, "3-D Shaded Computer Animation - Step-by-Step", *IEEE Computer Graphics and Applications* **3**, *pp. 18-25 (1983)*

4) Julian E. Gomez, TWIXT: A 3D Animation System. *Computers and Graphics* **9** *No. 3, pp. 291-298 (1985)*

5) Hanrahan, Pat and David Sturman, "Interactive Animation of Parametric Models", SIGGRAPH '85 Tutorial on *Introduction to Computer Animation* pp. 87-101 (August, 1985).

6) O'Donnell, T.J. and Arthur J. Olson, "GRAMPS - A Graphics Language Interpreter for Real-Time, Interactive, Three-Dimensional Picture Editing and Animation", *SIGGRAPH '81 Conference Proceedings*, **15** pp. 133-142 (July, 1981).

7) Reynolds, Craig W., "Computer Animation with Scripts and Actors", *SIGGRAPH '82 Conference Proceedings*, **16** pp. 313-322 (July, 1982).

8) Reynolds, Craig W., "Description and Control of Time and Dynamics in Computer Animation", SIGGRAPH '85 Tutorial on *Advanced Computer Graphics Animation* pp. 31-57 (August, 1985).

9) Shelley, Kim and Donald Greenburg, "Path Specification and Path Coherence", *SIGGRAPH '82 Conference Proceedings*, **16** pp. 157-166 (July, 1982).

Computer Animation with Scripts and Actors

by Craig W. Reynolds
Information International Inc.

Abstract

A technique and philosophy for controlling computer animation is discussed. Using the Actor/Scriptor Animation System (ASAS) a sequence is described by the animator as a formal written SCRIPT, which is in fact a program in an animation/graphic language. Getting the desired animation is then equivalent to "debugging" the script. Typical images manipulated with ASAS are synthetic, 3D perspective, color, shaded images. However, the animation control techniques are independent of the underlying software and hardware of the display system, so apply to other types (still, B&W, 2D, line drawing ...). Dynamic (and static) graphics are based on a set of geometric object data types and a set of geometric operators on these types. Both sets are extensible. The operators are applied to the objects under the control of modular animated program structures. These structures (called **actor**s) allow parallelism, independence, and optionally, synchronization, so that they can render the full range of the time sequencing of events. **Actor**s are the embodiment of imaginary players in a simulated movie. A type of animated number can be used to drive geometric expressions (nested geometrical operators) with dynamic parameters to produce animated objects. Ideas from programming styles used in current Artificial Intelligence research inspired the design of ASAS, which is in fact an extension to the Lisp programming environment. ASAS was developed in an academic research environment and made the transition to the "real world" of commercial motion graphics production.

CR Categories and Subject Descriptors: I.3 [**Computer Graphics**]: I.3.5 [**CG**]: Computational Geometry and Object Modeling; I.3.6 [**CG**]: Methodology and Techniques—*Languages*; I.3.7 [**CG**]: Three-Dimensional Graphics and Realism—*Animation*
General Terms: Design, Languages
Additional Key Words and Phrases: Lisp, Procedural Animation Languages, Motion Picture Production

Author's addresses:

US Mail:	III, 5933 Slauson Ave., Culver City, CA 90230
ARPAnet mail:	Reynolds@Rand-AI
Uucp network mail:	ucbvax!randvax!reynolds

Introduction

This paper describes the Actor/Scriptor Animation System (ASAS), which is a way of thinking about and describing computer graphic animation. ASAS is basically a *notation* for animated graphics. The notation for an animated sequence (the **script**) can be automatically read and converted into animated images by an ASAS interpreter. As in the case of musical notation being interpreted by a group of musicians—or the script of a video production being executed by a host of actors, camera, audio, lighting and video technicians—ASAS allows the creation and use of any number of simulated particpants, "**actor**s" each of which can control one or more aspects of the animation. The ability of ASAS **actor**s to operate independently or (by communicating with each other) to act in synchronization allows a simple and unambiguous description of the function of each **actor**.

ASAS differs from "performance" based real time computer graphics systems as well as from command or "menu" based systems. Writing the ASAS notation for an animated sequence will probably take longer than the final running time of the sequence. On the other hand, an ASAS **script** is typically more compact than a simple listing of the value of all relevant parameters for each frame, as might be required in a command-menu system. This results from the fact that ASAS is a procedural notation, a programming language for animation and graphics. In fact ASAS is a "full" programming language and includes all of the typical modern structured programming features (procedures (recursive), local variables, "if then else"s loops, typed data structures and generic operators). Additionally ASAS supports independent, parallel, "animated" program structures (**actor**s), and includes a rich set of geometric and photometric objects and generic operators on these objects.

The existence of a formal notation for a field of endeavor leads to a workable procedure for the development of an idea. Like an algorithm being debugged by a computer programmer, or a musical score being revised, an ASAS **script** being developed is both unambiguous and precisely modifiable. It is possible to change just one small aspect while keeping everything else exactly the same. This property of notation allows the process of progressive refinement ("tweaking") to be used to converge on the desired algorithm, music or animation.

History

ASAS was developed at the Architecture Machine Group at MIT as two thesis projects between 1975 and 1978 [24,24]. "ASAS 0" was not a full implementation, but "ASAS 1" did actually work,

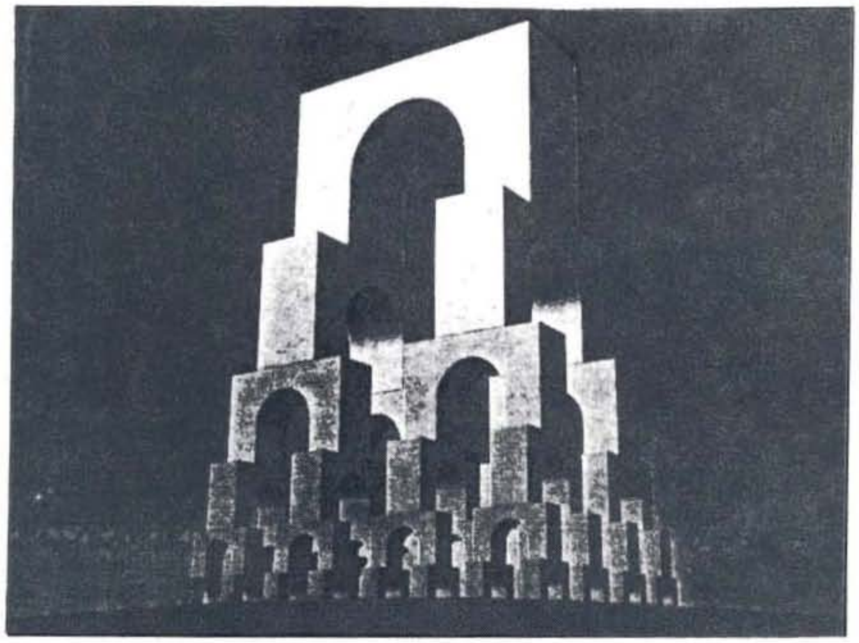


Figure 1: An Arch Fractal

despite a very slow and uninteresting display package. In 1979 ASAS was integrated into the Digital Scene Simulation system of Information International Inc. ("III", "triple-I") in Culver City, California. In this instance, ASAS is not used to make images directly, but serves as a preprocessor for III's existing 3D, hidden surface and shaded graphics system. Hence "ASAS 2" functioned as a true language compiler, translating from the animator's **script** to the command sequence for the display software. The inconvenience of having the display support in a separate software package is offset by the much wider range of graphic features made available to the ASAS user through the very advanced III software. After three years of commercial use the system was refined and became "ASAS 3". The current reference for the language is the unfinished **ASAS User's Manual 3.0**. [26]

The design of ASAS was influenced by some concepts from research in the Artificial Intelligence field. The basic concept of graphic databases and animation scripts as programs (procedural embedding of knowledge) was inspired by Terry Winograd's pioneering work in computer linguistics. In Winograd's system natural language was represented by a procedural data structure. [30] The concept of message passing **actors** was from Carl Hewitt's body of work in "actor systems" such as PLASMA. [12,13,14] (Similar concepts exist in Smalltalk [11], Simula [5] and Modula [32].)

An animation system in development at about the same time as ASAS by Ken Kahn shared some concepts with ASAS. [16,17] Kahn's system had what Hewitt calls a "uniform actor basis" and so perhaps a theoretically "cleaner" structure. Kahn's work placed more emphasis on embedding common-sense and theatrical knowledge in animation characters, and less emphasis on complex graphics.

ASAS and Lisp

As a programming language, ASAS "stands on the shoulders of giants" because of its relationship to the programming language Lisp. ASAS can be considered to be either implemented in, or an extension to, Lisp. The rich programming environment of ASAS is due largely to Lisp, since all Lisp primitives and utilities are usable from ASAS (and vice versa). A Lisp interpreter plus the ASAS software yields an ASAS interpreter. ASAS was developed under MagicSixLisp at The Architecture Machine Group at MIT and was painlessly transplanted to run under MIT MacLisp at III.

For a long time Lisp has had a reputation as somewhat of a "toy" language, a powerful but quaint tool used by gnomish academic artificial intelligence researchers, but not a language really suited for commercial use. These critisisms are misdirected, Lisp is one of the most elegant and useable algorithmic notations ever devised. The bad reputation is due mainly to poor implementations or underpowered computers. Because Lisp trades off raw computational efficiency for expressive power and usability, a well designed interpreter and a fast machine are required for a production environment. Today there are several good Lisp systems for various types of general purpose computers (MacLisp [20], InterLISP [29], ...) and currently three firms are selling specially designed Lisp machines (Lisp Machine Inc., Symbolics Inc., and Xerox)

ASAS Expressions

ASAS and Lisp use a simple, if unusual, notation. A "parenthesized prefix notation" is used for operators, control structures and data. An ASAS **expression** is either:

(1) a number
(2) a symbol ("variable")
(3) a parenthesized list of **expressions**.

If the expression is a list, the first (leftmost) thing is the name of an operator (or "function") and any other expressions in the list are parameters for the operator. When an **expression** is evaluated (or "executed"), numbers evaluate to themselves, symbols to their currently defined value, and a list (eg: "**(plus** a **(abs** b))") evaluates to the result of applying the operator ("**plus**") to the recursively evaluated parameters (values of "a" and "**(abs** b)"). For example, to define the symbol "wheels" to be the number of tricycles times 3, we would write:

```
(define wheels
        (times tricycles 3))
```

Normal ASAS operators (like **times**) evaluate each of their parameters, while certain operators have special evaluation patterns (like **define**, which does not evaluate its first parameter, these are called "macros"). To define a simple operator (call it "thrice") which multiplies its single parameter ("x") by three:

```
(defop thrice
       (param: x)
       (times x 3))
```

an equivalent definition in MacLisp would be:

```
        (defun thrice (x) (times x 3))
```

The first example could then be rewritten:

```
        (define wheels
                (thrice tricycles))
```

Special Symbols

Within a **script** certain aspects of the production are controlled by the values given to some special symbols. None of these symbols are actually "reserved words", but it is best to use the **script** symbols **background** and **camera** only for the purpose of defining the current **color** of the graphical background of the image and the current camera description (as a **pov**, see the section on geometric objects). The initial ASAS environment has other symbols defined to various frequently used objects (axes, colors, basic solids), it is good practice to know these and avoid redefining them.

Geometric Objects

In addition to the data types found in most programming languages, ASAS provides a set of geometric (and photometric) objects: **vector**, **color**, **polygon**, **solid**, **group**, **pov**, **subworld**, and **light**.

The **vector** represents a position in three dimensional Cartesian space. It allows three parameters, the X, Y and Z coordinates. Trailing zero coordinates may be omitted.

A **color** object may be specified either by its Red, Green and Blue components, or by Intensity, Hue, Saturation. The two operators are called **rgb** and **ihs**, each of which accept three numbers between 0 and 1.

A simple **polygon** contains a **color** and a list of **vector**s, the "boundary". The **cut-hole** operator allows the construction of **polygon**s with "holes and islands" (that is, multiple boundaries). The **color** can be a **group** of a front color and a back color. The boundary points may be listed separately or as a **group** of **vectors**. Here is a **polygon** expression for a certain blue triangle:

```
        (polygon blue
                (vector 1 0 0)
                (vector 0 1 0)
                (vector 0 0 1))
```

A **solid** represents a bounded region of space, a closed polyhedron. It is composed of vertices and faces (as **vector**s and **polygon**s) in addition to topological connection information.

Several geometric objects can be "glued together" into a **group** object, which is then manipulated as a whole by the geometric operators. A **group** expression allows any number of parameters -geometrical objects to be **group**ed together, including other **group**s.

```
        (define houses
                (group red-house yellow-house brown-house))
```

The "point of view" object (**pov**) is used to define the point of view of an observer (for example the ASAS **camera**) or of an object. That is, a **pov** describes the three coordinate axis basis vectors and the position of the origin of an arbitrary coordinate space. We refer to such spaces by names like "eye space" and "an object's local coordinate space". Note: a **pov** plays a role very similar to a "4X4 homogeneous transform matrix" in other 3D graphics systems (there is a simple transformation from a **pov** to a 4X4 matrix) but a **pov** is a geometrical object composed of **vectors** and can be manipulated just like any other object.

A **subworld** is an object associated with a **pov**. This allows ASAS to manipulate a complex object by modifying only the **pov**, hence various "instances" of an object may share the same underlying data. **Subworlds** also allow ASAS to work with "levels of abstraction" in a graphic database, when a **subworld** is formed it notes the "overall size" and "typical color" of its contents. At display time this allows efficient tree structured clipping (when an entire **subworld** is offscreen) and handling of detail too small to see (when an entire **subworld** lies within a single pixel). [4] Hence the user can build levels of abstraction

Figure 2: How to make an Arch Fractal (given an arch element).

```
(defop  arch-fractalizer
        (param:  arch-element top-color bot-color levels
                 fractal-ratio height width leg-width)
        (local:  (total-levels         levels)
                 (offset-dist          (half  (dif  width leg-width)))
                 (sub-tower-offset-1   (vector  offset-dist 0 0))
                 (sub-tower-offset-2   (mirror  x-axis sub-tower-offset-1)))
        (arch-tower  levels))

(defop  arch-tower
        (param:  levels)
        (if  (zerop  levels)
             (then  nothing)
             (else  (add-arch-level  (arch-tower  (dif   levels 1))))))

(defop  add-arch-level
        (param:  sub-tower)
        (grasp  sub-tower
                (scale  fractal-ratio)
                (move  (vector  0 height 0))
                (rotate  0.25 y-axis))
        (grasp  arch-element
                (recolor  (interp  (quo  levels total-levels)
                                   bot-color
                                   top-color)))
        (subworld  (group  arch-element
                           (move  subtower-offset-1 sub-tower)
                           (move  subtower-offset-2 sub-tower))))
```

into a geometric database by the nesting of **subworld** objects.

Objects to be seen in shaded images are illuminated by **light** objects. **Lights** are composed of a position **vector** and a **color**.

Geometric Operators

ASAS's geometric operators are the tools the animator uses to shape, move and orient objects. An object's shape may come directly from the action of operators, or parts encoded by hand with a digitizer can be assembled with the operators. The same operators are used both for static arrangements, or to create animated motion, by operating frame by frame under the control of an **actor**.

In many command/menu based graphics systems it is difficult to precisely specify the correct ordering of geometric transformation. For example, there will be a "rotate" command which accepts three numbers, the angles of rotation for each axis. Often there is no mention of in what order the rotations are applied, let alone a way to specify the desired order. In ASAS, the animator explictly determines the ordering of operations by the structure of the nesting of the expressions written in the **script**.

The basic operators are "generic", they can be given any type of geometric object and operate on it as is appropriate for that object's type. ASAS operators NEVER modify the object they are operating on. The value returned by an operator is a geometrically modified copy of the original object with otherwise the same type and structure.

A notational shorthand is provided for the common occurence of a series of operations to be performed on a single object. The object to be operated upon can be made the "current" object (using the **grasp** operator). The "**grasp**ed" object will then be redefined by calls to operators which do not explicitly specify an object to operate on.

Two basic types of geometric operators are provided by ASAS, "global" and "local" (sometimes called "self relative").The ASAS global geometric operators are called: **scale**, **move**, **rotate**, **stretch** and **mirror**.

Generally these operators apply the named geometrical transform to any given geometrical object The transforms are relative to the origin and major axes of the global coordinate space. The parameter types to each are numbers and **vector**s as appropriate. ("Stretch" is a differential scaling for each axis, specified by a **vector** of scale factors).

As an example of the usage of the ASAS global operators see Figures 1 and 2, these show how last year's SIGGRAPH cover was constructed.

The "local" operators are similar in effect to the global operators, except that they are based on an object's OWN coordinate system rather than the global coordinate system. A **subworld** carries along its own little coordinate system, its **pov**. Not only does this allow efficient modification of the **subworld** but it also provides a reference for operations in the object's local coordinate space. The local operators were inspired by the "turtle" of the LOGO graphics language [2], and are intended to be a three dimensional analog of the turtle operations (walk forwards or backwards, turn right or left). This notion of a 3D turtle (more of a deep sea swimming turtle than a land crawling tortoise) was first used by Jim Stansfield and then refined by Henry Lieberman in a 3D line drawing extension to LOGO. A good treatment of this subject can be found in [1]. Usually objects will be defined so that the origin of their local coordinate space is at the center of the object. For this reason we will informally refer to the "origin of the local coordinate system" as the "center". Local operators are provided for moving, rotating, scaling and "zooming" relative to the local coordinate system. All of these operators accept one or two parameters, the second optional parameter is the object to operate on, if none is specified, the currently **grasp**ed object is redefined.

Note: the relationship between global and local operators is similar to the that of pre- and post-multiplication of transform matrices. Also note: when objects other than **subworld**s or **pov**s are passed to self relative operators they are first put into an identity ("home") **subworld**, then operated on.
Local operators:

grow	scale up about local center
shrink	scale down about local center
forward	move along local +Z axis
backward	move along local -Z axis
left	rotate to left about local Y axis
right	rotate to right about local Y axis
up	rotate upward about local X axis
down	rotate downward about local X axis
cw	rotate clockwise about local Z axis
ccw	rotate counter-clockwise about local Z axis
zoom-in	scale up local Z axis
zoom-out	scale down local Z axis
local-move	move along arbitrary local vector
local-stretch	scale each local axis independently
home	resets back to original definition space

Examples: an operator sequence which (if evaluated each frame) will cause the AIRPLANE (that is, the sequence of objects which form the animated value of the variable AIRPLANE) to perform "barrel rolls", and one to cause the CAMERA to pan around while zooming out:

```
(grasp airplane)
(forward 0.1)
(cw 0.02)
(up 0.02)

(grasp camera)
(right pan-speed)
(zoom-out 1.01)
```

Various other ASAS operators are available but will not be discussed here. There are **recolor** and **cut-hole**, and **interp** the general purpose interpolater, **row** and **ring** which make regular **groups** of objects, and **prism** which makes solids by projecting a **polygon**. Here are some examples of some of them, an operator to make a n-sided regular polygon (inscribed within a unit radius circle), and an operator to make a prism with regular "ends":

```
(defop regular-polygon
       (param: color sides)

       (polygon color
                (ring sides
                      (vector 0 1 0)
                      z-axis)))

(defop regular-prism
       (param: color sides thickness)

       (prism color
              (vector 0 0 thickness)
              (regular-polygon color sides)))
```

This summary of ASAS operators suffers because of the language's extensibility; the full list is endless since the user invents new ones as needed. Beyond simple combinations of basic linear operations, there is a large class of nonlinear "bending" operators. For example consider "curl-up" which takes a long thin object and curls it into a spiral (Escher fans will know the application for that).

Scripts and Animate Blocks

The main program an ASAS user writes is called a **script**, which is a special type of **defop**. A **script** handles the setting up and setting down needed to produce an animated sequence (or write a file for later production by another system). The **script** expression includes a name and any number of subexpressions. The effect is to define an operator with that name which opens production, evaluates each expression in the body, and closes production. There is no restriction, but the things in the body are usually either **animate** expressions ("animate blocks") or production utilities (such as "make N blank frames", "put this slate text", or "make an N second countdown").

An **animate** block is a special type of loop. Each time around the loop, after it evaluates its body, a frame of animation is produced automatically. Usually the body contains **cue** expressions ("cue at frame N, ... "). These cause objects to be made visible (with the **see** operator) or start, stop or direct **actors**. **Animate** blocks are exited when a **cut** operator is evaluated.

This example **script** contains one **animate** block, which starts two similar **actors** at different times. Both **actors** then run until the end of the block.

```
(script spinning-cubes

        (local: (runtime 96)
                (midpoint (half runtime)))

        (animate (cue (at 0)
                      (start (spin-cube-actor green)))

                 (cue (at midpoint)
                      (start (spin-cube-actor blue)))

                 (cue (at runtime)
                      (cut))))
```

(Note: **cut** accepts an optional frame number, and will cut only if that is the current frame, so that third **cue** could have been written as "**(cut runtime)**".) When an **animate** block is exited, all of the **actors** associated with it are stopped. Hence **animate** blocks are somewhat like the "scenes" of a movie, the coarse structure of the action.

While an ASAS "cue" is in fact simply a number (a frame number relative to the current **animate** block) it should not be thought of as a constant. Because of the computational nature of a **script** it can be quite easy to move cues around, since all cue points can be handled symbolically (by name rather than by a literal number). For example it is a simple matter to change the overall runtime of a **script** (for a "quick run through" test) if all cue points are defined relative to one variable (e.g. "runtime"). Library macros exist to facilitate just such a scheme. The animator may find it necessary for artistic reasons to move a cue point within a **script**, again this will be quite painless if everything which is supposed to begin or end at that cue point refers to it only symbolically.

Actors

The control structure of an animation system would be very simple if we could assume that all sequences to be produced had at most one independently animated feature at any one time. On the other hand, if we assume that there may be any number of fully independent animated features (starting and stopping at random times, happening at different rates, running in sync or not) then conventional control structures are no longer the most appropriate.

An ASAS **actor** can be thought of several ways. Most basically an **actor** is a "chunk" of code which will be executed once each frame. Usually an **actor** (or a team of them) is responsible for one visible element in an animation sequence, hence it contains all values and computations which relate to that object. In this sense an **actor** serves to modularize and localize the code related to one aspect, isolating it from unrelated code. From a formal point of view, an **actor** is an independent computing process in a non-hierarchical system with synchronized activation and able to communicate with other **actors** by message passing.

When an **actor** is "on" (between being **start**ed and **stop**ed), it will be awakened once each frame, its local variables restored, its body evaluated, its variables saved, then put back to sleep. (Hence an **actor** has properties between a "closure" and a "process" in recent Lisp implementations.) **Actors** are put into action with the **start** operator, which takes an **actor** and returns the "actor instance id", a unique number for each active **actor**. An **actor** can deactivate itself, or can be gunned down by the **script** or another **actor**, this is done with the **stop** operator which accepts an "actor instance id". **Run** is a combination of **start** and **stop**, it starts an **actor** with a predetermined stop cue. This is the definition of the operator "spin-cube-actor" used in the **script** "spinning-cubes":

```
(defop spin-cube-actor
       (param: color)

       (actor (local: (angle 0)
                      (d-angle (quo 3 runtime))
                      (my-cube (recolor color cube)))

              (see (rotate angle y-axis my-cube))

              (define angle
                      (plus angle d-angle))))
```

It expects one parameter, a **color**, and returns an **actor** object.

The **actor** itself has three local variables, each of which is assigned an initial value in this case: "angle" is the current angle of rotation for this **actor**s cube, "d-angle" is the incremental velocity of the angle, "my-cube" is a recolored version of ASAS's predefined "cube" **solid**. Each frame the **actor** constructs the rotated version of "my-cube" and passes it to the displayer, the current "angle" is updated for the next frame.

Animated Numbers

In the last example the symbol "angle" took on a series of numeric values, frame by frame, forming an arithmetic series. But for more complex time behavior (quadratic or cubic curves) the inline code to handle and update all those linear difference terms becomes a burden. To avoid this, ASAS supports an animated numeric object called a **newton** (as in Newtonian mechanics). **Newton**s can be used any place a number would be used, such as a coordinate in a **vector** or the angle parameter for **rotate**. Between frames however, **newton**s are automatically updated to the next value in their predefined sequence. The **newton** data structure holds its future as a chain of piecewise cubic curves with selectable degree of continuity at the joints.

A **newton** can be specified in terms of position, velocity, acceleration and delta acceleration ("jerk" or "jerkiness") when those values are known. But more typically **newton**s are defined with utilities which produce curves with certain properties. Animators are familiar with terms like "slow in" or "slow out" meaning that an action should start (or end) with zero velocity (first derivative). The five most common curves (or pieces of curves) used in ASAS are: **hold, linear, slowin, slowout** and **slowio**. **Hold** accepts a value and a length of time, each of the others takes a starting and ending value and a time. **Slowio** (slow in and slow out) has zero derivatives at both ends. When none of the standard curves are appropriate an interpolating cubic spline fit is used.

Actors and Behavior Simulation

Some animation is made to match a preconceived image, especially in commercial production. Other times, animation is produced as an experiment, the answer to "what would happen if ...". In the second type, which might be called "behavior simulation", the animator sets up a little world by defining the rules of behavior and selecting the cast of characters. When the behavior simulation is run we obtain images of what went on in the little world.

A classic example of this sort of thing is to try to build a computer graphic simulation of a flock of birds. We must define the behavior of a single bird so that when a lot of instances of the bird are simulated, they flock convincingly. The flock seems to be following a leader, but each time they turn, a new bird becomes the leader. The flock changes direction like a single unit, yet it is just an assembly of individuals. The flock is a dense cluster, but the birds do not often collide.

ASAS **actor**s provide a convenient way of implementing such behavior simulations. As mentioned before, one of the features of **actor**s is the way they promote "separation of powers", independent modules of code which do not interfere with each other. This allows an **actor** to take the part of one characters in the simulation. If the same sort of character occurs many times in the simulation (e.g. many copies of BIRD) we can use independent "instances" of a given "class" of **actor**s.

The other key feature of **actor**s which makes them suitable for behavior simulation is the ability to pass **messages**. Clearly the birds in the flock are exchanging information, through the action of light and sound on the bird's senses each one is aware of where the others are and where they are going (in an intrinsically depth sorted order!) In an **actor** simulation of the flock we would not go to the extent of modeling light and sound, but we could realistically have each bird broadcasting to everyone the message "I am here (x y z) and I'm heading (dx dy dz)". In that implementation, each bird would have to put the others in order of importance, probably using a "hidden bird algorithm".

Message Passing

ASAS **messages** are handled by two special operators: **send** and **receive**. **Send** composes messages and posts them at the recipient's mailbox. **Receive** reads each message in the mailbox, responding to each in a manner depending on the type of message. (ASAS **actor**s act once each frame, not whenever a message comes, hence mail may pile up between frames so the mailbox is implemented as a FIFO queue.)

Send takes an address, a "message type" (optional), and any other specific message data (numbers, geometric objects, symbols). The address is either an **actor id** or a special symbol: "**all**" means send to all actors and "**script**" and "**animate**" can be used to send messages to the surrounding **script** or **animate** block. The "message type" is any symbol used to describe the type of message, it must match the message type in the recipient's **receive** construct. For example, these **send**s (1) tell "bouncer" to speedup by 10 percent and (2) announce to all birds where we are:

```
(send  bouncer speedup 1.10)

(send  all bird-state cur-position cur-velocity)
```

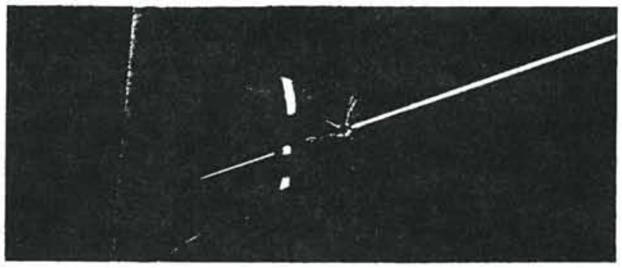
The **receive** construct has a body much like a **case** construct. Each message in the mailbox is examined in turn, the "message type" of each is compared with the type of the various clauses. If one of the clauses in the body of the **receive** matches the incoming message type, the body of the clause is evaluated in response to the message. Message type "**any**" in a clause will match any incoming message. The contents of a message (past the type) may be accessed by specifying parameter bindings for the clause type. For example, this **actor** knows how to receive only "speedup" and "slowdown" message:

```
(receive ((speedup f)     (define speed
                                  (times speed f)))

          ((slowdown f)   (define speed
                                  (quo speed f)))

          (any            (print  'What?)))
```

The message passing mechanism described above is based on a more primative operator called **In**. The **In** operator allows the evaluation of any expression "inside" the local variable space of an **actor**, thus allowing examining and setting the local variables of the **actor**. This is a useful but dangerous tool which should be used only in a well designed protocol.

Commercial Production at III

ASAS frequently plays a central role in commercial animation production at III, although other techniques of animation control are used. Projects made with ASAS include, "MICROMA" animated logo, "LBS" animated logo, "NEWS CENTER 2" TV news show intro, two TV commercials for "TORNADO", various magazine ads, all of the theme animation for the III 1981 Sample Reel ("The Juggler"), about half of the special effects for the Ladd Company's feature motion picture "LOOKER", and all of the animation and still images III is making for the recently released Disney feature "TRON".



Copyright © 1982 by Walt Disney Productions and Information International, Inc.
Figure 3: Solar Sailer Escape Sequence from TRON

The Digital Scene Simulation Group usually works on a contract basis with clients such as film and video producers and advertising agencies. Some projects come to us carefully planned out by the client, while others come in a very vague form. If we do not get specific artistic directions (timings, storyboards and renderings) from the client, our Art Department creates these materials in consultation with the client. A team of at least three staff members (art director, designer/encoder, technical director) is formed to work on the job.

When the artistic concept is somewhat settled, work is started on its computer graphic realization. The first step is to create a geometrical model of the shapes of the objects to be used in the animation. Unless the geometry of the object is regular enough to allow it to be constructed under program control, the shape definition is done by hand in a laborious process similar to technnical drafting we call "encoding". Often some mix of manual data entry and processing by various "geometrical tool" programs is used to obtain a finished object shape description.

As the objects are being finalized, the technical director begins to write the ASAS scripts and related programs. When the group is working many closely related scenes, for instance during production on a feature film, many of the "sets" or environments will be shared between several scenes. In such cases it becomes convenient to set up "libraries" of common ASAS function and object definitions. Contributions are made to these "public libraries" by all animators working on a project. Usually the motions in an animated sequence are so specifically planned out in advance that an outline of the script can be written before any of the objects are available for test pictures. At this stage the ASAS script is very abstract, references are made to symbolic constants whose values are not yet known. When object files are ready and the script is roughed out the "graphical debugging" begins.

Often design constraints are stated in such an indirect fashion ("have the camera pointing such that the logo is positioned here and oriented like in this sketch") that the only workable way to find the desired numerical parameters is experimentally with graphical feedback. After specific "key" frames have been composed, and the transitions between them defined, a motion test is made. Usually this test is made in either line ("vector") or low resolution shaded image mode. The symbolic nature of ASAS scripts make it easy to adjust the runtime of a sequence, making preliminary tests at 10:1 speed ratios allows faster turn around. Also the script can be simplified for these tests, by dropping out certain elements, replacing others with "stand-ins", all these changes can be made under the control of ASAS script flags. Often the motion test will reveal problems in the "feel" of the dynamics of the animation or unexpected behavior between the key frames.

Another pass or two is made to finalize the motion and then attention is shifted to the color, lighting, shading and other "photometeric" parameters of the animation. Key frames are examined on a high resolution video display and color tests are made onto the type of color film which will be used for the final image. Often a parameter will be determined with a "wedge test", making a series of frames which differ only by a single parameter value (e.g. the amount of ambient light in the scene), with the gamut of values laid out, the final value can be easily be selected.

When all has been decided, a high resolution final filming is made. Typically when the result is screened, the client will find at least one reason to reject the work and the whole process goes back to the beginning.

Conclusion

This paper has presented ASAS, a general purpose programming language which has been extended to include geometric objects and operators, parallel control structures and other features to make it useful for animated computer graphic applications. ASAS makes use of an abstract computing element called an **actor**, we have seen how **actor**s promote modularity and how they can simulate a wide range of behavior by exchanging messages with each other. In three years of commercial use ASAS has proved itself a workable and practical tool. While the specific feature a user wants may not already be a part of the language, the extensibility of ASAS allows it to grow with its users. ASAS has expanded our "complexity barrier" another notch, allowing us to attempt work with more independently animated elements than before.

The author is not prepared to state that when the ULTIMATE computer animation system is built, it will be programming language based. But it is hard to visualize a system which allows arbitrary extensions into unexpected realms without being fully programmable. However, programming and making aesthetic judgements seem to be disjoint in most people's thinking processes. The user of a graphics programming system must always be on guard against compromising aesthetic judgements to simplify the programming! The solution used in our commercial work is to make the production a joint effort of several people, some responsible for artistic issues and others responsible for technical issues.

References

[1] Abelson, H. and diDessa, A. *Turtle Geometry*, MIT Press (Series in Artificial Intelligence), Cambridge, MA, 1981.

[2] Austin, H., "The LOGO Primer", MIT A.I. Lab. Logo Working Paper 19.

[3] Church, A "The Calculi of Lambda Conversions", *Annals of Mathematical Studies* 6, Princeton University Press 1941, Reprinted by Klaus Reprint Co., 1965.

[4] Clark, J., "Hierarchical Geometric Models for Visible Surface Algorithms", *CACM*, October 1976.

[5] Dahl, Myhrhaug, and Nygaard *The SIMULA 67 Common Base Language*, Norwegian Computing Centre, Oslo, 1968.

[6] DeFanti, T. "The Digital Component of the Circle Graphics Habitat", *Proceedings NCC 1976*.

[7] Dijkstra, E.W. "Notes on Structured Programming", August 1969

[8] Eastman, C. and Henrion, M. "GLIDE: A Language for Design Information Systems", *SIGGRAPH '77 Proceedings*, July 1977, San Jose, CA.

[9] Futrelle, R. P. and Barta, G. "Towards the Design of an Intrinsically Graphical Language", *SIGGRAPH '78 Proceedings*, August 1978, Atlanta, GA.

[10] Goates, G., Griss, M. and Herron, G. "PICTUREBALM: A Lisp-based Graphics Language System with Flexible Syntax and Hierarchical Data Structure", *SIGGRAPH '80 Proceedings*, July 1980, Seattle, WA.

[11] Goldberg, A. and Kay, A. *SMALLTALK-72 Instruction Manual* Learning Research Group, Xerox Palo Alto Research, March 1976.

[12] Greif, I. and Hewitt, C. "Actor Semantics of PLANNER-73", *Proc. of ACM SIGPLAN-SIGACT Conf.*, Palo Alto, CA, January 1975.

[13] Hewitt, C. and Smith, B. "Towards a Programming Apprentice", MIT AI Lab Working Paper 90, January 1975.

[14] Hewitt, C. and Atkinson, R., "Parallelism and Synchronization in Actor System", *ACM Symposium on Principles of Programming Languages 4*, January 1977, L. A. CA.

[15] Jones, B. "An extended ALGOL-60 for Shaded Computer Graphics", *ACM SIGPLAN/SIGGRAPH Symposium on Graphical Languages*, April 1976.

[16] Kahn, K. "An Actor-Based Computer Animation Language", *Proc. of the ACM-SIGGRAPH Workshop on User-Oriented Design of Computer Graphics Systems*, Pittsburg, PA, October 1976.

[17] Kahn, K., "A Computational Theory Of Animation", MIT A.I. Lab. Working Paper 145, April 1977.

[18] Larkin, F. "Computing with Text-Graphic Forms", *Conference Record of the 1980 LISP Conference*, August 1980, Stanford University.

[19] Larkin, F. "A Structure from Manipulation for Text-Graphic Objects", *SIGGRAPH '80 Proceedings*, July 1980, Seattle, WA.

[20] Moon, D. *MacLisp Reference Manual, Revision 0*, MIT Project MAC, December 1975.

[21] Newman, W. and Sproull, R. *Principles of Interactive Computer Graphics*, McGraw-Hill, 1973 and 1979.

[22] Pfister, G. "A High Level Language Extension for Creating and Controlling Dynamic Pictures", *ACM SIGPLAN/SIGGRAPH Symposium on Graphical Languages*, April 1976.

[23] Preissler, M. "Multi-processed Music Synthesis", BS Thesis MIT EECS Department, May 1976.

[24] Reynolds, C. "A Multiprocessing Approach to Computer Animation", SB thesis, MIT EECS Department, August 1975.

[25] Reynolds, C. "Computer Animation in the World of Actors and Scripts", SM thesis, MIT (Architecture Machine Group), May 1978

[26] Reynolds, C. *Actor / Scriptor Animation System User's Manual 3.0, (ASASUM 3)*, Information International Inc., in preparation.

[27] Smoliar, S. "A Parallel Processing Model of Musical Structures" MIT AI Lab Technical Report 242, September 1971.

[28] Sussman, G. and Steele, G. "SCHEME - An Interpreter for Extended Lambda Calculus", MIT AI Lab Memo 349, December 1975.

[29] Teitelman, W. *InterLISP Reference Manual*, Xerox Palo Alto Research Center, 1978.

[30] Winograd, T. *Understanding Natural Language*, Academic Press, 1974.

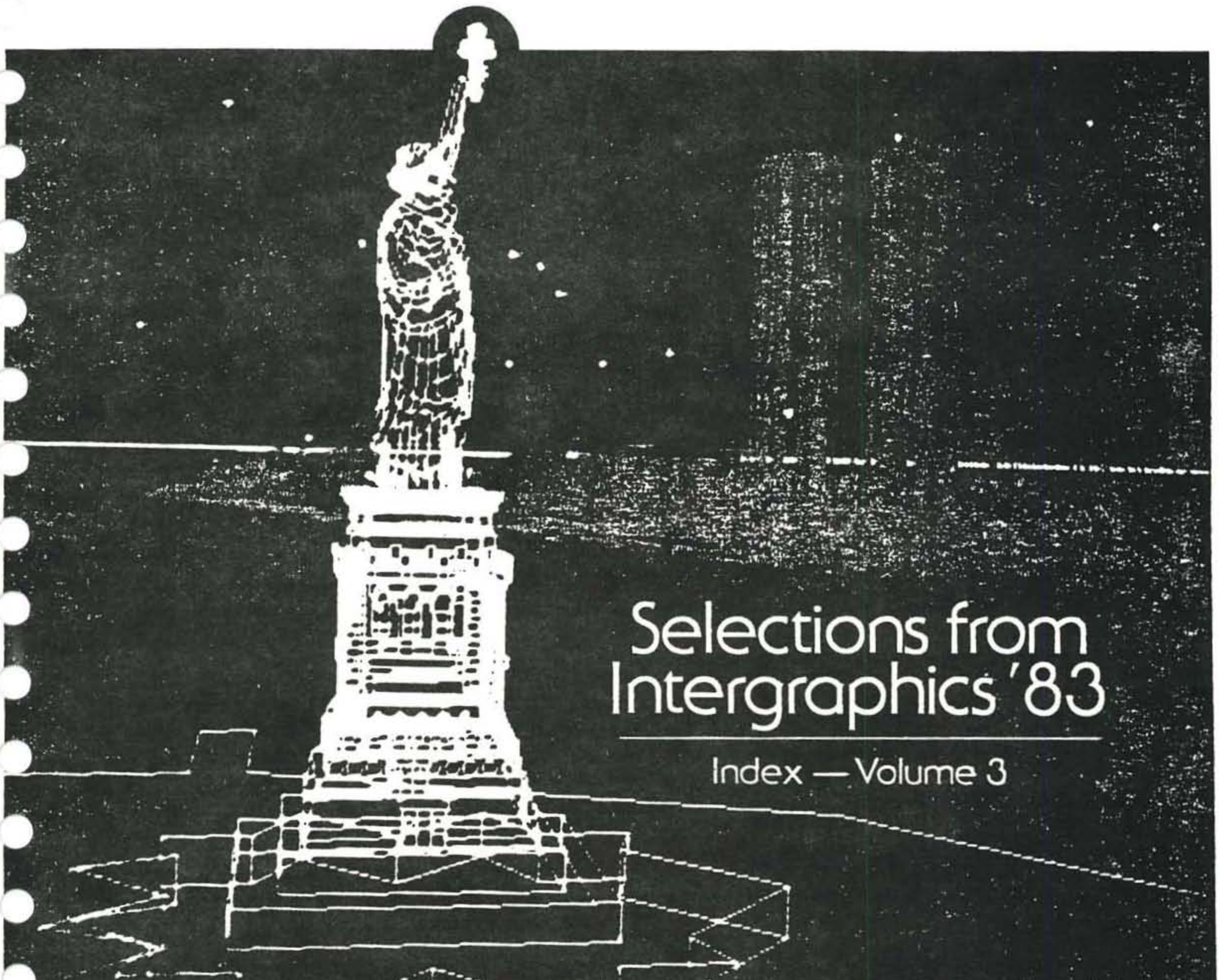
[31] Winston, P. and Horn, B. *Lisp*, Addison Wesley, 1981.

[32] Wirth, N. "MODULA: a Language for Modular Multiprogramming", *Software, Practice and Experience* 7,1; 1977 pp. 3-35.

Note

This entire paper is an example of computer graphics. All pictures were produced with Digital Scene Simulation, and directly (digitally) converted into four color halftones in the Infocolor format.The text was edited and composed on TECS, and assembled with a Page Makeup System. Camera ready, full page art (including typesetting and halftone generation) was produced with a COMp80/2 Pagesetter. All of these systems are products of Information International Inc.

IEEE
Computer Graphics and applications



Selections from Intergraphics '83

Index — Volume 3

DECEMBER 1983

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

National Computer Graphics Association

A walk through the production process at this commercial CG animation house reveals an emphasis on software tools and the use of intermediate-stage graphics for design flexibility.

3-D Shaded Computer Animation—Step by Step

Richard Chuang and Glenn Entis
Pacific Data Images

The animation of computer-generated images has made a number of contributions to the entertainment industry, and the future of these images in film and television looks promising. Although many computer animation techniques have been widely researched and published, their actual commercial use in production studios is still in its relative infancy.

The creation of animation for the entertainment field requires a combination of technical capability, design sense, and practical production skills. In this article, we describe the use of computer graphics techniques in a production environment; in doing so, we stress the practice, rather than theory, of computer graphics.

Pacific Data Images produces commercial computer-generated animation for broadcast television. What follows is a step-by-step explanation of how commercial animation is produced at our facility. In the course of this explanation, we will discuss several tools we've built to support the animation process. In particular, we emphasize software at PDI, since software continues to be the key technical animation component that cannot be bought "off-the-shelf."

The techniques covered in this article provide an example of how a typical piece is produced in our studio, so we make no claim of universality. Nonetheless, this walk through our production process should impart a sense of what producing computer animation is like.

System overview. The PDI animation system was developed specifically for the production of commercial animation. It produces smooth animation of 3-D shaded raster images and features an animation language, a modeling tool kit, real-time animation design, and several viewing options. Output is to 16mm or 35mm film, or to one-inch video tape.

Figure 1 presents an overview of our animation system. Animation in our system is defined by scripts "written" in our animation language, by world coordinate polygon files, and by motion data. This data is processed by the script program, which interprets the animation script and creates one of several types of polygonal output files.

Screen coordinate polygon files can be sent either to the renderer for the production of antialiased, shaded raster images, or to a wireframe display program that displays the polygon edges in antialiased vector form. World coordinate polygon files can also be displayed in wireframe form and can define models used by other scripts. Vector files are sent to the real-time vector device for interactive viewing and animation design.

Images are displayed on color monitors, vector terminals, and the system's real-time vector device; images can also be sent directly to one of two film recorders or to an NTSC encoder for recording onto videotape. The color monitors are located at graphics workstations, which are equipped with terminals and data tablets for interactive use by animation designers.

The script system

We rely heavily on a script system for our animation design. Our script system is a special-purpose graphics language supporting animation at a high level. The script system was influenced by a number of other, similar systems, most notably Craig Reynold's Lisp-based ASAS.[1] Our script is built on top of the C programming language[2] and thus shares many of C's features. We designed our script for simple syntax; default values for the entire graphic environment; support of complex modeling, transformations, and motion; and support of modular scripting. The data structures and file types used in the script are shared with other PDI design tools and, thus, provide a common ground on which all design programs communicate.

We use the name "script" because of the way the animation language is used in the production process. In the early stages of a job, the basic models and motion for an animation are roughed out in the script language. This original script already includes elementary timing, lighting, viewing, and modeling information that will later be refined and used in the construction of the final product. At each production stage, this script is updated to reflect production changes and to incorporate new models and motion data from other parts of the system. Thus, the script is much like a movie script in that it concisely describes *what* should happen *when* and *where* in the production. Unlike a movie script, however, the script also defines the animation and is run directly to create both the intermediate and final versions of a job.

There are several advantages to this approach. The script was built to be a fast prototyping language for image building and motion testing. Typically, the preliminary rough script for an animation sequence is built in the very first session with a client. This capability for rapid prototyping is essential for design experimentation and flexibility.

Next, the script directly handles data types and modeling operations for lighting, viewing, and transformation, thus freeing the scriptwriter to concentrate on *what* has to be done, rather than *how* it has to be implemented. Most script values default to predefined values if not explicitly set by the animator, making simple object definition fast and easy. For example, if a five-sided prism is required, the script command is

```
prism 5
```

Unspecified prism attributes, such as color, radius, height, and surface type, all default to standard and/or previously set values. In addition, viewing parameters, such as camera position, focal length, lights, window, and viewport, are all set in the script (or, again, default to standard values) so that when an object is created, it can be transformed into its final screen position.

Since the animation for a sequence is originally prototyped and refined in the same script that creates the final piece, the possibility for error and the necessity of tedious data transcription is greatly reduced.

Motion design

Since the main task of the director and animator in 3-D animation is the choreography of objects and camera, we developed a motion design system to assist them in this. Our motion design system consists of (1) several interactive and noninteractive motion design tools that set up each scene and (2) a number of alternative methods for viewing each animation sequence. Figure 2 illustrates the data flow during the motion design process.

The interactive motion design system utilizes a vector display to establish key-frame positions for the camera

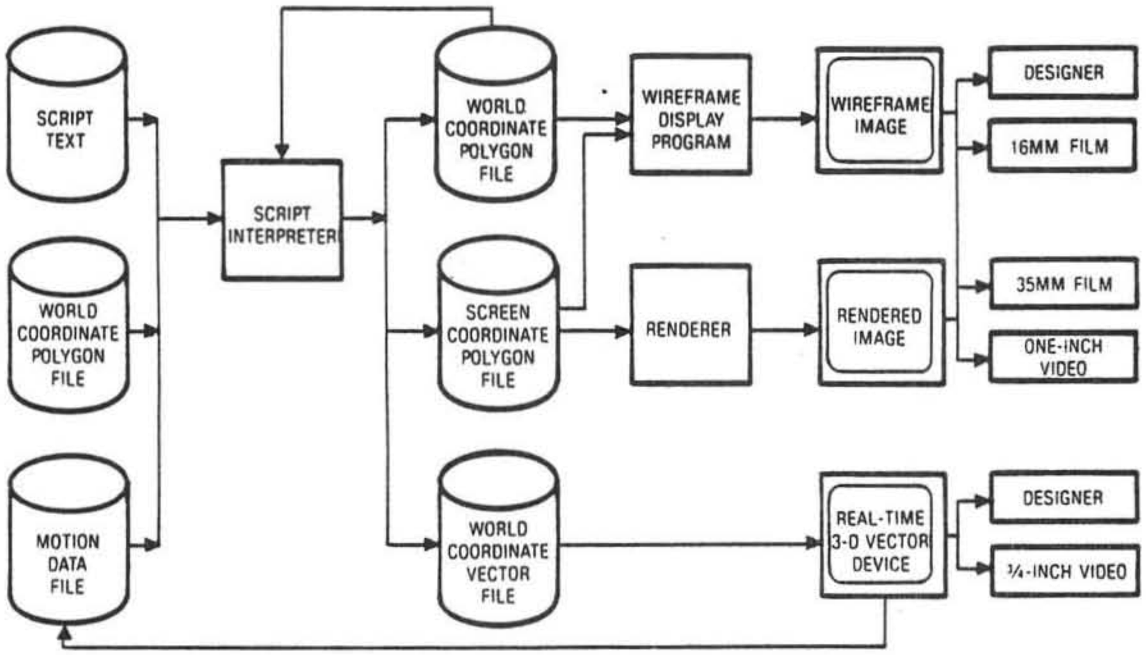

SCRIPT TEXT
WORLD COORDINATE POLYGON FILE
MOTION DATA FILE
SCRIPT INTERPRETER
WORLD COORDINATE POLYGON FILE
SCREEN COORDINATE POLYGON FILE
WORLD COORDINATE VECTOR FILE
WIREFRAME DISPLAY PROGRAM
RENDERER
WIREFRAME IMAGE
RENDERED IMAGE
REAL-TIME 3-D VECTOR DEVICE
DESIGNER
16MM FILM
35MM FILM
ONE-INCH VIDEO
DESIGNER
¾-INCH VIDEO

Figure 1. Overview of the Pacific Data Images animation system.

and actors. The camera's view, focal length, and direction of view, along with the position and orientation of each actor, are specified by the animator interactively. The key frames are then placed into an animation sequence. Next, as specified by the animator, the computer generates the required number of positions for all the intermediate frames.

Mathematical control paths for both space and time can also be used to modify the transition of motions between key frames. Examples of key-frame modification are camera tracking, easing in, easing out, and roll and pitch. For well-defined motions, numerical paths can be entered manually or interactively. All motion specifications can interact within the script of each animated scene. Combining numerical paths and key-frame specifications is often used in our animation.

Extensive use of motion paths simplifies the design tools needed for specifying complex motions. By using a simple description of motion in terms of motion paths, new animators can quickly learn the basics of 3-D computer animation and later extend their knowledge to more complex scene choreography.

Previewing motion. An animated sequence can be previewed in several ways. First, a flip-book approach can help an animator view his design in a somewhat "traditional" manner. In this approach, all motions in an animation sequence are calculated and drive the motion on the real-time 3-D vector device. This device can play the animation back at any speed, which allows the animator to "flip" through the sequence at varying rates. Having access to this level of interactive control is important for an animator in that it helps him understand the quality of movement and permits him to closely (and slowly) scrutinize movement.

A second preview method involves filming a wireframe pencil test of an animated sequence. This produces a display of the actual choreography. Through the use of very simple models or "stand-ins," motion tests can be done prior to having the actual actors in each scene.

Then, finally, the motion can be previewed at low resolution (256 × 243). Here, the choreography of movement, lighting, and color are seen together before the final rendering. This low-resolution animation is visually more helpful to an art director than vector animation, since a vector animation sequence contains no information as to the color and mass of each scene.

Model design

A designer has control over many characteristics of the objects to be animated, including geometric shape and size,[3,4] color, surface quality,[5] and amount of detail. As described above, very simple models are sometimes used in previewing a sequence so that the designer can get a feel for the overall scene or movement within a scene. At other stages of production, different details of the models are required to complete and fine-tune the final piece. All of this falls under the general heading of model design.

An animator may be called on to animate almost any kind of object, so it is important that his available modeling tools be flexible and fast. A prime objective of our modeling tools is to provide just such a set of general capabilities—one that allows quick object definition without the necessity of using custom software.

Figure 3 illustrates the various tools that, together, create a finished model. The script language, various modeling programs, and a graphics editor each produce a common-format world coordinate polygon file. This world file can, in turn, be processed by other modeling programs. Additionally, the file can be further manipulated by the script language. Once an object's world

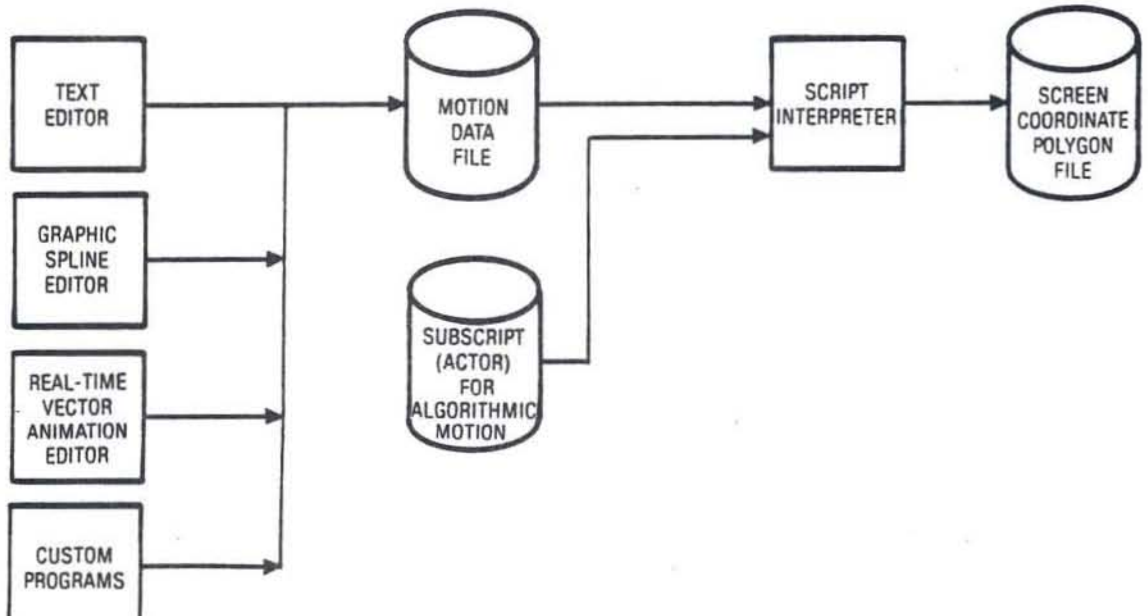

TEXT EDITOR
GRAPHIC SPLINE EDITOR
REAL-TIME VECTOR ANIMATION EDITOR
CUSTOM PROGRAMS
MOTION DATA FILE
SUBSCRIPT (ACTOR) FOR ALGORITHMIC MOTION
SCRIPT INTERPRETER
SCREEN COORDINATE POLYGON FILE

Figure 2. Motion data flowchart.

polygon file is completed, it can be called up by the script controlling the animation for conversion into screen coordinates and then incorporated into the final scene. Each of these steps is examined in detail below.

The script language described above is used to create most models based on geometric primitives and to build up parts of models created by other programs. Quickly specified simple objects made from primitives are especially useful as stand-in objects. But the script is the basis of both the animation and modeling systems, so all script primitives can also be used as building blocks for more complicated models. Many models, such as buildings with multiple levels, windows, and doors, can be constructed by combining very simple geometric forms (see Figure 4).

Other modeling programs create various types of spline-based objects, including patches, free-form surfaces, and "swirls." There is also a special program for creating three-dimensional logos from two-dimensional contours. The splines and contours used by these programs can be entered in three ways: (1) using an interactive spline editor, (2) from spline data created in a text editor, or (3) from other programs. Figures 5 and 6 illustrate models defined with various spline-based surfaces.

Still other programs form various geometric surfaces on polygonal meshes. For example, a recent job called for the animation of rocky surfaces, so a fractal modeler[6] was used to create mountains.

We also use a special class of modeling programs based on a Unix concept known as "filters." A filter program is one that accepts a stream of data, performs some transformation on that data, and outputs the data, usually in its original, input format. The advantage of this approach is that each filter program can be designed to do one very specific task and do it well. Complex tasks are then performed by chaining filter programs so that the output from one program is the input to the next. This type of "combinatorial power" provides very extensive graphical capabilities with relatively simple software. Our filter programs accept world coordinate polygon files as input, process them in some way (alter polygon normals, change colors, apply nonlinear transforma-

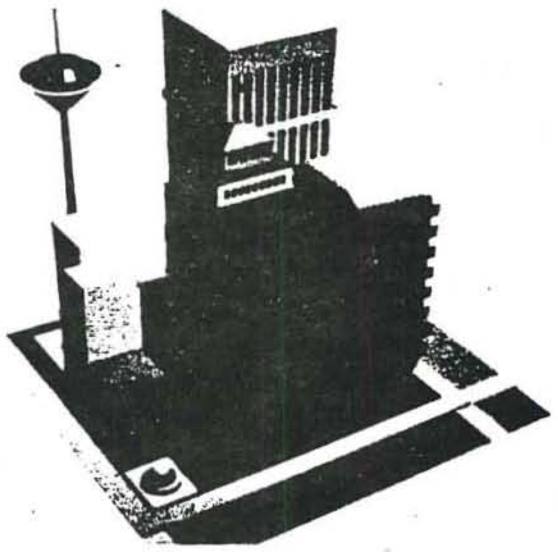


Figure 4. All elements of this scene are simple geometric primitives created by the script.

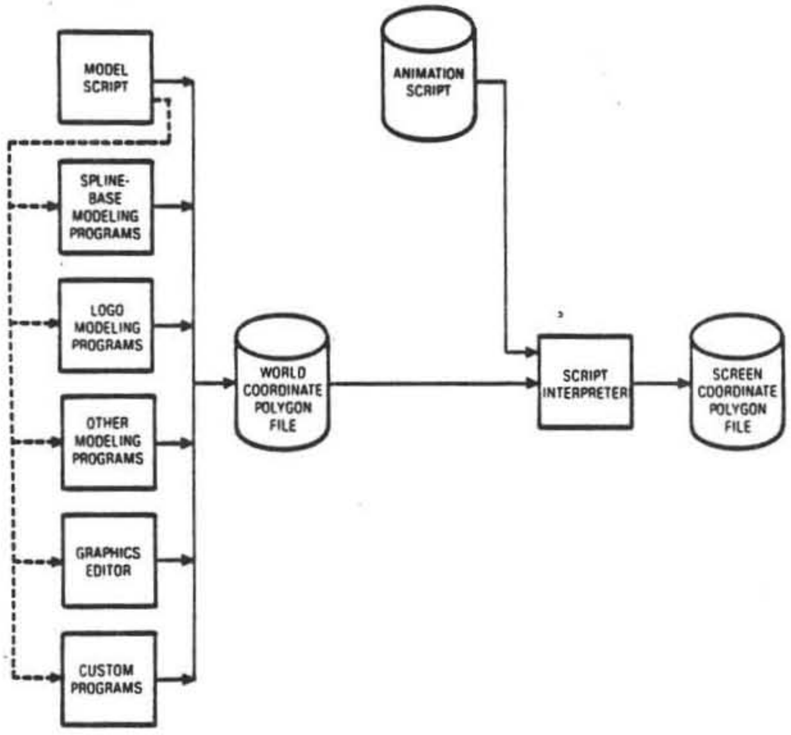


Figure 3. Both model scripts and animation scripts are written in the PDI script language. Note that the script can also be used to drive the other modeling tools.

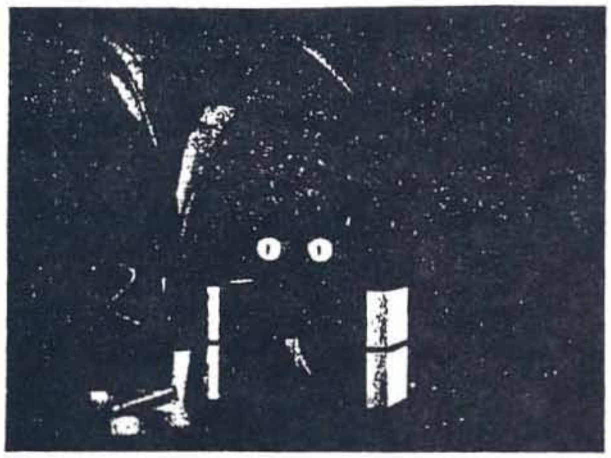


Figure 5. Cat Clock's trim, plant leaves, smoke, and ashtray were all built from B-spline-based objects.

Figure 6. This image features stochastically generated spline-based models, textured ground, and special effects glow.

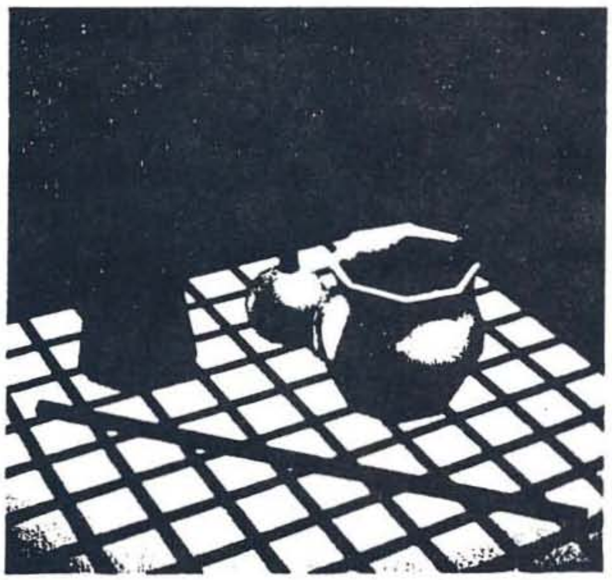


Figure 7. Stand-in model of a still life.

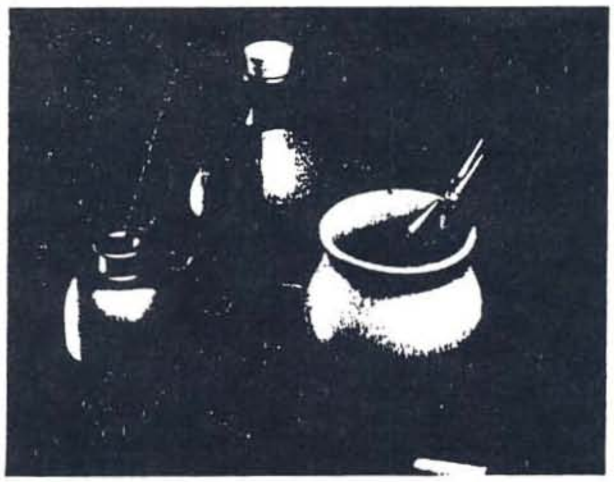


Figure 8. Detailed rendering of a still life.

tions, etc.), and then output a new world coordinate polygon file.

When a geometric or algorithmic modeling approach won't suffice, an interactive graphics editor is also available. As the artist draws the model with a data tablet and interactive display, the computer stores the artist's drawing in a file. This information is converted into polygons, which can then be used as a model or as input into other modeling programs. Many of our logos and our more free-form designs are modeled in just this fashion. In some cases, the most practical method of building a model is by scanning the original art into the display memory with a video camera, displaying the image on the interactive display as a guide, and then tracing the scanned-in image to create the final model. Once defined, this final model is available for display and animation from the script with a single command.

Rendering a computer graphics model, which is done after the model has been defined, means displaying it in color on a CRT screen as a solid object with hidden surfaces removed and proper 3-D shading. Stand-in models are again useful—this time for the determination of color, object placement in the scene, and scale. An example of a scene of rendered stand-in models (polygonal stand-ins in a still-life) is shown in Figure 7. The colors and final placement of the objects were determined using those stand-ins, after which the final, detailed scene (Figure 8) was created.

Designing a complicated object correctly on the first attempt is difficult, so a fast method of viewing objects in the design state is desirable. In our system, any model can be previewed as a color wireframe (Figure 9). Such wireframes have the advantage of being quick to display and yet still show the full structure of the scene. Wireframe drawings can also be plotted on paper for detailed planning and documentation.

Once created, models are rarely thrown away. Most computer animation studios build up the equivalent of a shopping catalog of models that they can use in future work. Our script system encourages the modular construction of models from subparts. Each model is then given a name and saved in a model library. If a scene is made of several models, the script retrieves the necessary

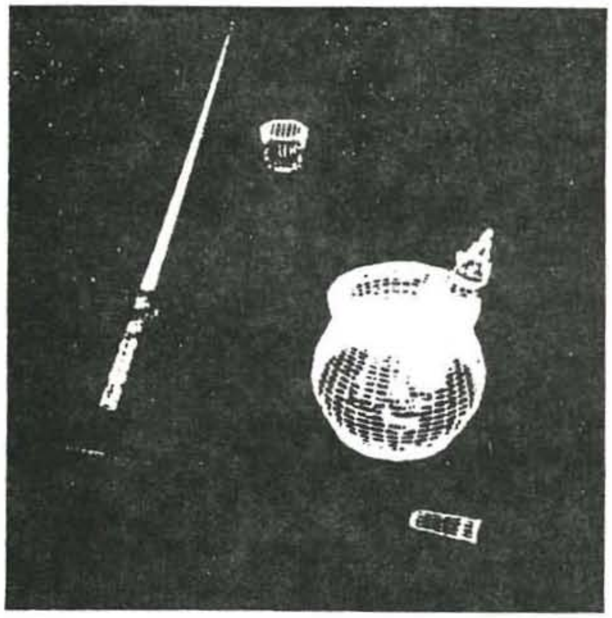


Figure 9. Color wireframe of a still life.

models from the library. Because each model is independently defined, the script can be used to isolate objects for documentation and detailed viewing.

After the model has been fully rendered by the designer, it can be placed at any location on the screen, either by itself or in combination with other models. Additionally, the camera can be positioned to aim at any screen location. A virtually unlimited number of light sources, at any position and of any color, are allowed; in fact, at this stage of design, a major portion of the artist's and client's time is spent adjusting the lighting and fine-tuning the color of the model. Subtle lighting effects, highlighting, and color balance can be adjusted to greatly enhance the strength of the final image.

Test shots

Models and animated movements can be used any number of times and in a variety of ways. This is particularly advantageous in preparing test shots; a script can be progressively refined and viewed in inexpensive test-shot formats until everyone agrees that they have what they want.

The above section on motion design mentioned the use of flip-book motion previewing and wireframe models. When combined, these two techniques offer a relatively cheap method of previewing the animation for an entire production.

In traditional animation, a *pencil test* is the filmed sequence of raw, pencil art shown at the normal animated playback speed. For the computer graphics equivalent of pencil tests, we use the script and models for the final piece, but draw each model as a color wireframe rather than as a fully rendered image. A wireframe image can be up to several orders of magnitude faster to compute than a fully rendered image, yet the wireframe pencil test still displays the motion exactly as it will appear in the final

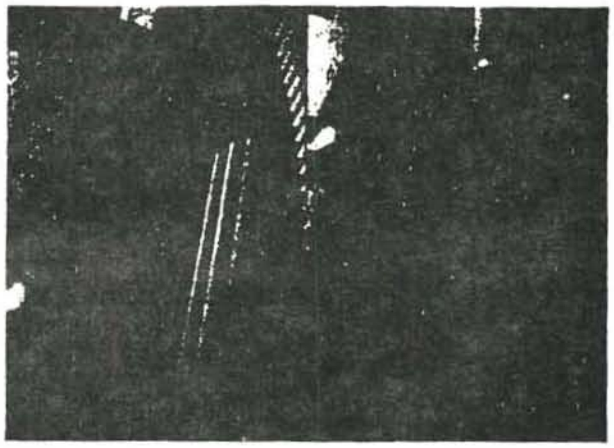


Figure 10. This composite image shows the same antialiased scene at 512 × 486, 256 × 243, and 128 × 121 resolution.

work. Because the same script and models are used for the pencil test and for the final piece, the movement and models in the test are duplicatable. Thus, misunderstandings and organizational overhead in the studio are reduced.

Test renderings are made at various stages of the animation so that the designer can view and modify the look of the models, coloring, lighting, and the general feel of the key frames. Often a scene consists of many individual elements, including mattes for mixing live action with animation. Since test renderings, pencil tests, and the final animation are all made from the same script and models, it is a simple matter to choose any frame from the pencil test and render it exactly as it will appear in the final image.

Usually, the last piece of test footage in a normal production cycle is a low-resolution rendered animation sequence. This test has hidden surfaces removed, 3-D shading, and is antialiased, but all at lower resolution than what would be considered normally acceptable for a finished piece. Rendering frames at one-quarter resolution (reduced to half resolution, both vertically and horizontally) gives everybody involved in a production an excellent feel for the final film. Since resolution is closely tied to computing time, the low-resolution test shot is still significantly cheaper to compute than the final image. And in most cases low-resolution tests are done at one-sixteenth final image size (reduced to one-quarter size, both vertically and horizontally). Figure 10 shows a composite of an image generated at 512 × 486-, 256 × 243-, and 128 × 121-pixel resolution.

Rendered test shots need to be done at the final frame rate. In commercial productions, special lighting effects often take place in a very short time interval and are synchronized to a sound track. Seeing each of the final frames will assure the designer that all of his planned effects are in place and working. To make certain this happens, all the motion data has to be gathered and updated before doing the test shot. Figure 11 shows how all of the disparate pieces of information are coordinated to form the final image.

Important to all test shots is the ease with which they can be produced. In some cases, a motion test or rendered frame is available in a few minutes time, and since everything is repeatable, the animation designer has precise control over what is to be changed and what remains the same. As a parallel, imagine an architect who can refine a set of blueprints, build scale models from the blueprints, and then automatically have the building constructed directly from those blueprints.

The final shot

After all necessary test shots have been completed and everyone concerned with the production is satisfied with them, the final animation sequence is created. In our studio, this final product is shot on either 16mm or 35mm film through the use of a computer-controlled film recorder.

We generally prefer to produce final animation sequences directly onto one-inch videotape. To do this, we use a "director's" language to control the frame rendering of the final scene from the animation script and also to automatically control the film recorder. If multiple scripts are used, or if the scene requires images from outside sources to be matted with the computer-synthesized images, the director's language also handles this. Additionally, various special effects, such as glows and vectors, are added to the composite image at this point.

Because rendering a single frame of a complex image can take a long time, each element of a scene must be updated and synchronized before the final animation is recorded. The completed spot then goes to the client, and all scripts, models, and selected intermediate images from the animation are archived in a database.

The above is a general overview of one approach to the production of computer animation. The tools used in our studio and by the industry in general are new and improving rapidly. Ultimately, the quality of an animation sequence is the direct result of the quality of the graphic design and animation sense that goes into it. Good technical tools support animation design by providing clean, simple, and powerful methods of executing a design concept.

The tool-kit approach outlined here emphasizes general animation capabilities rather than custom software. This simplifies the software management task and gives animation designers a clear idea of what is possible. Our studio has been open for commercial production since May 1983, and we have completed almost a dozen commercial productions since that time. About half of these jobs were performed without the use of custom software, while the remainder required very simple custom programs that were generalized and documented, and thus became a standard capability of the studio. ■

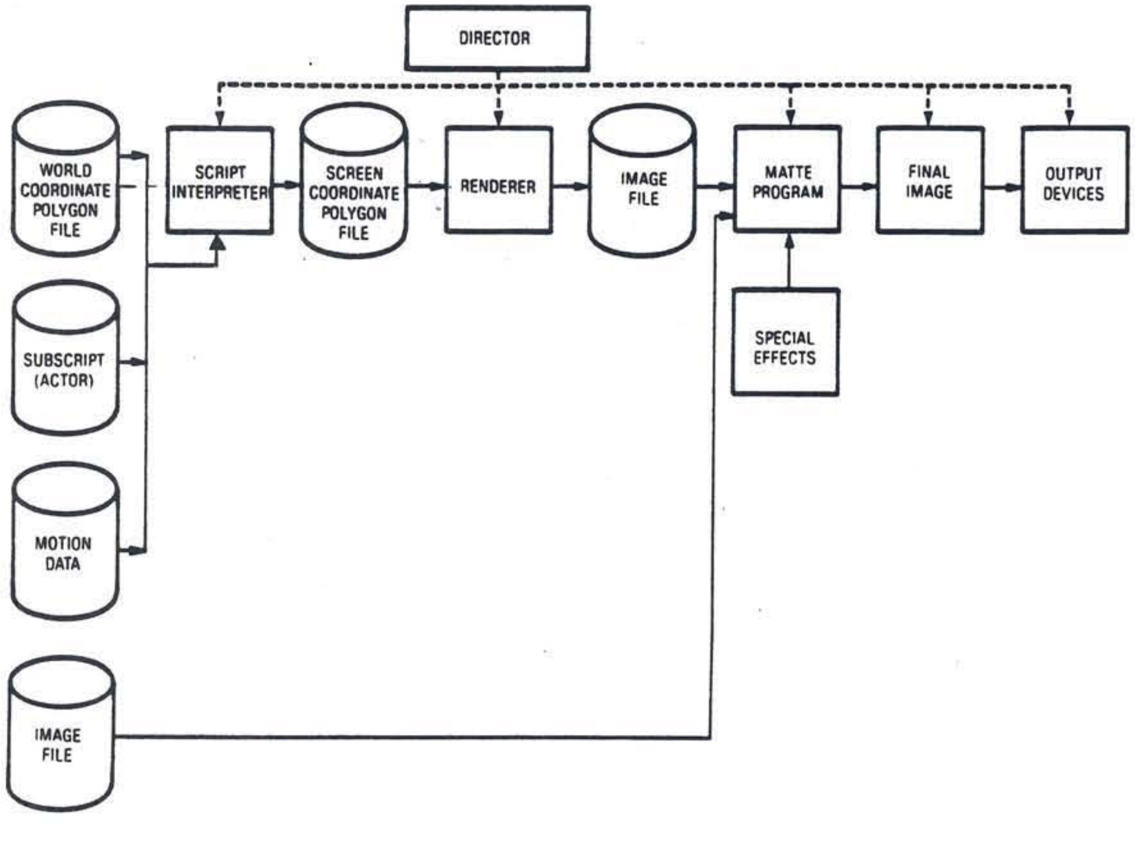


Figure 11. Data coordination required to produce a final image.

Appendix

Our current studio at Pacific Data Images was formed early in 1982. The objective of our three-person group was to produce computer-generated animation for broadcast television and film. In this we feel we have been successful.

Our present animation system has been in development since the studio was formed and was completely designed and implemented in-house. The basic system was built around a small minicomputer, a PDP 11/44, and a 512 × 512 × 32-bit DeAnza IP6400 frame buffer. At the end of 1982, we added a VAX11/750 minicomputer system, a DeAnza IP8500 frame buffer, and an IMI 500 real-time vector display station to our facility; in September 1983, we obtained a Ridge 32 supermini.

As stated earlier, all of our software was written in C under the Unix operating environment. Our selection of Unix and C was dictated by a desire for smooth development and growth. We also use low-resolution raster graphic terminals at our desks for design and preview.

One of our primary goals has always been to develop a state-of-the-art animation system capable of accommodating growth. The ability of the software to be transported to newer and faster computers is essential to the development of a productive and competitive animation capability, and modular design is critical to developing a transforming system. As new animation and image synthesis techniques become available, they will be integrated into our animation system with well-defined interaction.

For more information about our equipment and motion design, see the article by Rosendahl.[7]

Richard Chuang, vice-president of hardware engineering and technical director at Pacific Data Images, was one of the original designers of PDI's animation system. From 1979-1982, he was a member of the technical staff at Hewlett-Packard's RF and microwave division. His research interests include computer animation system architecture, advance rendering techniques, and design tools. He received a BSEE from the University of California, Davis, in 1979.

Glenn Entis is vice-president of software and technical director at PDI and was one of the developers of the PDI animation system. Prior to joining PDI, he was a software engineer at Hewlett-Packard and at Ampex where he worked on the AVA paint system.

Entis graduated from Ohio Wesleyan University with a BA in philosophy and a BFA in fine arts in 1976 and attended graduate school in computer science at the Polytechnic Institute of New York. He is a member of the IEEE Computer Society and the ACM.

References

1. Craig Reynolds, "Computer Animation with Scripts and Actors," *Computer Graphics* (Proc. Siggraph '82), Vol. 16, No. 3, July 1982, pp. 289-296.
2. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. William Newman and Robert F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, Hightstown, N.J., 1979.
4. James D. Foley and Andries van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982.
5. Robert L. Cook and Kenneth Torrance, "A Reflectance Model for Computer Graphics," *Computer Graphics* (Proc. Siggraph '81), Vol. 15, No. 3, Aug. 1981, pp. 307-316.
6. Alain Fournier, Don Fussell, and Loren Carpenter, "Computer Rendering of Stochastic Models," *Comm. ACM*, Vol. 25, No. 6, June 1982, pp. 371-384.
7. Carl O. Rosendahl, "Designing for Computer Animation," *Computer Graphics World*, Vol. 6, No. 10, Oct. 1983, pp. 38-40.

Comput. & Graphics Vol. 9, No. 3, pp. 291–298, 1985
Printed in Great Britain.

0097-8493/85 $3.00 + .00
© 1985 Pergamon Press Ltd.

Eurographics '84 Award Paper

TWIXT: A 3D ANIMATION SYSTEM†‡

JULIAN E. GÓMEZ§
Computer Graphics Research Group, The Ohio State University, Columbus, OH 43201, U.S.A.

Abstract—This paper describes a visually interactive 3D computer animation system. Animation is controlled by lists of events stored in tracks. Tracks are interpolated by functions that have an arbitrary degree of knowledge about how to control display parameters. The system provides real time wireframe playback so the animator can see what the animation looks like and is both input and output device independent.

1. INTRODUCTION

Computer animation is an important and popular topic which has received a great deal of attention recently. Many 2D and cartoon animation systems have been implemented (e.g., [1, 3, 4]) to interpolate, or *inbetween* monochrome and color pictures. However, these systems are animation assistants rather than animators, because they require conventional animators to draw frames that are close to the final image.

We are interested in three-dimensional animation, where the animator can create an alternate reality and manipulate the objects within it. A number of 3D animation systems (e.g., [5, 6, 9, 15, 19, 21]) have been implemented and are in current use. These systems take various approaches to the problem of designing a three-dimensional animation, which typically involve programming in a high-level language. This is a major disadvantage, in that the animator must learn computer programming before being able to animate. Without the appropriate high-level constructs, this means that the animator must furthermore learn how to associate the high-level concepts of animation with the low-level operations provided by a programming language.

In addition, because of the complexities of evaluating the programmed motion, these systems provide only *offline* playback, in that the animator must begin a computation sequence and return at some later point in time to view the sequence. This process may also require hardcopying the sequence (i.e. film or video) before being able to view it.

In this paper we describe *twixt*, a 3D animation system in use at the Computer Graphics Research Group of the Ohio State University. *twixt* is an interactive animation design system where the animator interacts visually. *twixt* has been used to generate a number of animated sequences, including *The Uneven Bars* [18].

The next section of this paper presents the design criteria for *twixt*. The remaining two sections discuss animation design and some major implementation features.

2. DESIGN CRITERIA

The design criteria in this section are discussed both in terms of how useful they are and how they are implemented.

2.1 *Visual interaction*

Our first criterion is that the animator interact visually. Many recent 3D animation systems [5, 8, 10, 15, 13, 21] require text mediated interaction, in that the animator must edit a file containing some kind of program in a high-level language that describes the animation. The animators have to wait various lengths of time for frames to compute before being able to see them, causing the animation process to be something of a trial and error affair. We specify that in *twixt* the animator works interactively with images and will see the image changing dynamically as graphical input is processed.

2.2 *Multiple track*

Our next criterion has to do with the specification of animation. We did not want to implement a *keyframe* system, in which the animation system knows everything there is at one particular frame (the *keyframe*) and interpolates for inbetween frames. Ideally, a keyframe is necessary only at a transition point, since the system can interpolate the rest. Unfortunately, a keyframe records one state, the total state of the frame. Some objects in the scene may not have any transitions at that frame, indicating that a keyframe is necessary wherever any object has a key action, leading to a large number of keyframes. Thus objects that don't really belong in a keyframe must be pre-interpolated before capturing the keyframe.

We can think of each object in an animation as pursuing its own course of action; therefore we should allow the animator to create *tracks* of action and place objects on their own tracks. The animator specifies what the object is doing at various points in time along those tracks. At those points we capture the display parameters of the object. For intermediate times we mathematically interpolate those parameters to con-

† The research described in this paper was supported, in part, by National Science Foundation grants MCS 79-20977 and MCS 79-23670.

‡ This paper is one of the three award winning papers from EUROGRAPHICS '84, the annual Conference of the European Association for Computer Graphics. The paper is published here in a revised form, with permission of North Holland Publishing Company (Amsterdam), the Publisher of EUROGRAPHICS '84 Conference Proceedings.

§ Author's address: CGRG/Cranston Center, 1501 Neil Ave., Columbus, OH 43201, USA.

struct *inbetween* frames. A related earlier effort is MUTAN [9], which allows the animator to place objects on their own tracks and specify synchronization marks where different tracks should line up.

As it turns out, one display parameter of an object could easily be independent of another, so we need a more complex approach: We must provide tracks for *each* display parameter of an object, not the object as a whole. For example, something might be spinning around while it jumps up and down, and we should allow each of those motions to have its own track. On these more specific tracks we will have *events* whenever something happens. We will interpolate between events to find inbetween values.

Events on any particular track are necessary only when the track begins to change its behavior. How tracks are evaluated to produce animation will be discussed later.

2.3 *Metamorphosis*

One of an object's tracks is its surface definition. In *twixt*, the surface definition of the animator's data as it's drawn on the screen is time dependent. We can interpolate between static data definitions to produce a third dataset, the one which is actually drawn. This will be discussed more later.

2.4 *Real-time playback*

Conventional animation techniques incorporate filmed rough sketches of animations in progress to get an idea of what the animations look like before going through the laborious process of inking and painting each frame. The rough sketches are drawn with pencils, giving the name *pencil test* to the film footage of the rough sketch. This is related to the *cartoons* used in classical painting.

Computation of the color version of an animation is still quite expensive. *twixt* provides pencil test playback in real time (or as close to it as a multiuser system will allow) so the animator can see what the motion and composition of the animation look like before spending a large amount of time on computation.

The real-time playback criterion is important, as an animator cannot finalize any sequences before seeing what they actually look like. *twixt* incorporates various computational tricks to speed calculation of inbetween frames; these will be discussed later.

2.5 *Device independence*

twixt operation is not limited to a particular display device; this point needs no discussion. *twixt* can run from anywhere in the laboratory, not just from a graphics workstation. If the animator is fortunate enough to be using a graphics workstation, *twixt* provides visual feedback, and the animator has graphical input devices with which to interact. Otherwise, the animator still has access to *twixt*'s power, but picture generation has to be done on terminals with vector graphics capability, with no real time interaction.

2.6 *CGRG animation environment*

One of the final, but certainly important criteria, is that *twixt* fit into the environment at CGRG. This strongly affects its user interface, in that commands must be designed in a way that corresponds with the rest of the programs at CGRG, and that the way the animator handles data corresponds with the rest of the graphics software. For example, if the animator asks to see the color of something, *twixt* produces the same report as any other software in the animation pipeline. *twixt*'s design has influenced the design of other software in the animation system, and other programs have borrowed ideas and code from *twixt* for their operation.

This criterion also involves access to high quality imagery with a minimum of effort. To draw images in color, *twixt* speaks directly to *scn assmblr* [7], a system designed by Frank Crow for varying quality image generation. The animator at any time can generate a high quality image of the current working frame with a simple command (see example below). This capability extends to allow the animator to take an animation designed with *twixt* and have it generate the equivalent command file for later offline computation of the high-quality imagery (again with one command).

2.7 *Pecuniary paucity*

Finally, the reason for some design decisions was the lack of available resources. This is a common complaint among research institutions, and is often the final arbiter in many decisions, as it was in some cases here.

3. ANIMATION

The unit of time in *twixt* is the frame number. This proceeds at 24 frames per second for cinema and 30 for video. It can be argued that it would make more sense to deal with animation in terms of seconds. In any case, the notion of frame is historically tied into animation, and would be as difficult to overcome as switching from the foot to the meter.

Animation in *twixt* is not limited to moving things around in space. Animation in *twixt* is dynamically changing display parameters, which can include values outside the transformation matrix, like color. Thus objects like light sources can have dynamic color, which will not be seen until the frame is computed on a color display.

3.1 *Events and tracks*

As previously discussed, each object has a number of display parameters. Whenever one of these changes value, we designate it an *event*, and enter it into a linked list of events for that particular parameter. Besides its frame number, an event also has other parameters associated with it (e.g., ease-in count, linear interpolation, Catmull–Rom interpolation, etc.).

The event list is known as a *track*. Tracks are doubly linked [12] because a program can go forwards or backwards in the list in the same amount of time, and quite rapidly. When interpolating, it is necessary to access both values preceding and succeeding the current value, and these accesses must be done very quickly in order to maintain real time response. This implementation scheme sacrifices space for time.

Commands are available to shift the frame number of events (thereby changing their timing) and the functions associated with the event.

The collection of all tracks is known as the *script*. The script is maintained in editable form on the file system, so the animator can look at it or even perform external modifications on it. The script is separate from the intermediate form that *twixt* transmits to *scn assmblr*. As with text editors, it is a good idea to write out the script every once in a while in case the system crashes (or the animator finds a fatal bug in *twixt*).

3.2 *Interpolation*

There are a plethora of techniques for interpolating or approximating curves between points; see [16] for a start.

3.2.1 *Linear.* Much animation is done with linear interpolation, using

$$c = c_0 + \alpha * (c_1 - c_0)$$

modified to perform sine curve acceleration/deceleration. We explicitly allow α to range outside [0.0 + 1.0] in order to extrapolate as well as interpolate.

Since we must access values within events, we define an abstract function

$$E\text{ (object, event, parameter)},$$

where *object* indicates what object, *event* points to the event in question, and *parameter* indicates the parameter for which the value is desired. E can be boolean valued, scalar valued, vector valued, matrix valued, an interpolation function, etc. Then the linear interpolation function becomes (as appropriate):

$$\text{temp} = E\text{ (object, } \epsilon_0, \psi)$$
$$r = \text{temp} + \alpha * \times (E\text{ (object, } \epsilon_1, \psi) - \text{temp}).$$

Acceleration and deceleration in classical animation are known, respectively, as *ease-in* and *ease-out*, conveying the idea that an object eases into a movement from rest and eases out from movement to rest. This can be generalized to acceleration and deceleration. We can provide acceleration by modifying the interpolation technique to proceed nonlinearly along the interpolated curve.

Conventionally, eased movement is done with a circular interpolation technique amounting to a portion of the sine curve. This portion ranges from $-\pi/2$ to $+\pi/2$ for acceleration, and $\pi/2$ to $3\pi/2$ for deceleration. In between ease-ins and ease-outs (either or both of which may be zero length) linear motion is used.

3.2.2 *Second degree.* A couple of second-degree interpolation techniques are useful. One is to fit a circular section through three points (after handling the collinear special case). Another is to use blended parabolas, or Overhauser interpolation [2, 16].

3.2.3 *Third degree.* Cubic splines have a rich literature, having been studied a great deal in mathematics and CAD. We provide some of the common splines to aid the animator in fitting smooth curves through display parameters.

3.3 *Frame construction*

To build a frame, *twixt* evaluates the activity on every track of every object. Once the display parameters for an object have been determined, its matrix is built, various flags are set, and static data definitions are interpolated if necessary. Thus the frame can be considered as the union of activity on all tracks.

This approach is important because it allows the animator to specify different motions independently and have *twixt* take care of putting them all together. Using the previous example (which is exactly delineated in the appendix) the animator can design the spinning motion, then the jumping motion, and the two sequences are added together inherently.

3.4 *Abstract track control*

The preceding sections dealt with the basic movement capabilities of *twixt*, but it is possible to define higher-level functions. A first step would be to link tracks together, so that one display parameter is controlled by a function applied to another display parameter. A good example would be to make the position of a rolling object be correctly determined from its angular velocity (or vice versa), or the velocity of two objects being controlled by their distance from each other.

A basic version of this capability is the ability to transform tracks: Once a track has been set up, it can be copied and transformed to another track of equivalent type (i.e. scalar, boolean, vector, etc.). An example would be to take object *A*'s position and multiply it by -1 into object *B*'s position. The resulting animation would have *B* exactly mirroring *A*'s position.

This feature allows the animator to use *twixt* at varying levels between a guiding system and an animator system [20]. The animator is no longer limited to designing absolute motion; a track becomes a static piece of data which can be instanced to produce actual animation. In other words, animation designed in *twixt* can be used as procedural animation as well as actual animation.

In general, evaluation of a track requires evaluation of some function, of which those mentioned above are only a few examples. There are two constraints for a movement function: It should be first-order continuous (C^1) so that the object does not suddenly do anything (like change position drastically), and it should be second-order continuous (C^2) so that the velocities involved in displaying the object do not change suddenly.

It is important to think of an event in an abstract sense. It is not limited to scalars or vectors or even the quantities mentioned previously. It is better to think of a track as an n-vector that provides the interpolation function with enough information about a certain point of time that the function can compute necessary values for an animation. In the common case of 3-vectors interpolated with a cubic spline, this is a straightforward operation. A more complex function, for example an n-body problem, could compute the position of n bodies, the final result for each body ending up on its position track.

The idea of using a movement function is very general, since complex functions will eventually generate specific values for an object's display parameters. The animator has a distinct advantage when he can feed parameters to high-level functions and let them take care of compilations down to the track level.

4. USER INTERFACE

This section presents some of the major implementation points about *twixt*. A detailed explanation would not be in order here; the disscussion is limited primarily to data use and manipulation.

4.1 *Frame editing*

In a basic sense, we can consider *twixt* a picture editor. The illustrator can simply work with *twixt* to design a still image, interactively adjusting the scene until it looks right, generating color versions intermittently.

In this mode, the illustrator probably won't invoke any animation functions. We say "probably," because the ability to vary an image over time provides the illustrator with yet another interactive tool in designing the image. By animating aspects of the composition, many variations can be generated rapidly and automatically. The illustrator can then "pick out" desired portions of a frame description.

4.2 *Space vs time*

A fundamental approach in the implementation of *twixt* is to sacrifice space for time. The real-time constraint requires maintenance of a large number of variables: By profligate storage of state and transition variables, we can avoid a large amount of computation. With a modern virtual memory computer (such as a VAX), the time over space sacrifice is no big deal, and for the benefits makes a lot of sense.

4.3 *Objects*

4.3.1 *Modeling technique.* The overwhelming majority of surface modeling at the Computer Graphics Research Group is done with polygon meshes, although *twixt* will also handle various kinds of bicubic patches.

4.3.2 *Object creation. twixt* works with *instances* [14] of data defined previously by the animator and resident in the host filesystem. There are a few different ways of using instances in *twixt*. The first simply displays the polygons (or patches) of the data. An edge dictionary (for polygonal data) is created on the fly and sent to the display device. The second method is to display points for each vertex of the data, rather than edges: it is used for high complexity objects which otherwise would use up too much display device memory. A variation on this method for bicubic data is to draw their control point mesh. The third method does not display anything, but keeps the surface data around for later interpolation, i.e. a *definer*.

Once an instance has been created, we include it in an *object*. An object includes the instance just created, and all static and dynamic parameters required to display it, such as color, position, orientation, etc. An object may be a *blender* (as opposed to a regular object) which means that the animator provides a list of definers and frame numbers at which to use each of those definers. The object is drawn by interpolating between the appropriate definers (thus each definition is an event on the blender's surface definition track). Note that the set of definers is reusable among different blenders.

4.3.3 *Special objects.* Some special objects are provided by *twixt* for the animator: the virtual eyepoint, the center of interest, the background, one default light (up to 16 can be requested), and ambient light factor. All of these can be animated.

The eyepoint and center of interest, along with the view angle and roll angle, model the virtual camera that is viewing the scene. The background can be either a solid wash of color or a precomputed image. If it is a color, the color can be animated; if it is an image, it can be either static or dynamic. A dynamic image background would be the situation where the animator has precomputed an animation, and now wishes to use those scenes as backgrounds for new action.

4.3.4 *Hierarchies.* We implement hierarchical objects by *attaching* one object to another [7]. This defines a general tree creating one complex object from sets of objects. Thus a scene in *twixt* is actually a *forest* [12] of hierarchical objects.

It is important to realize that although some objects have special meanings, they are still objects. Thus the eyepoint or center of interest can be attached just like any other object. To implement an *unfixed truck*, where the camera moves with a moving actor while following that actor, the animator would attach the center of interest to the actor, and attach the eyepoint to the actor at the proper camera displacement.

4.3.5 *Naming.* There are times when the animator will want to speak of an object as whole, and there are times when just one value of that object is desired. There are also intermediate stages, i.e. when the animator desires a major chunk of an object. It is necessary to provide a naming syntax that will handle all of these cases.

We took an approach fashioned after most ALGOL-based languages: A parameter of an object is addressed by *objectname.field.* To access more than one field in an object, the animator concatenates: *objectname.field1\field* 2 . . . A null field name is equivalent to naming the whole object.

The term *field* is ambiguous, since a field can be a boolean, a scalar, a vector, a matrix, a surface definition, or some other primitive used to implement a display parameter. An object's position, for example, is a vector, but the *X* coordinate of that vector is a scalar. These two quantities would be named, respectively, *objectname.p* and *objectname.px*.

4.4 *Terminal interface*

4.4.1 *Commands.* Commands in *twixt* follow the same format that seems to be generally accepted for interactive programs and systems. The syntax is like a simplified UNIX shell [17]: a word (the command) optionally followed by a set of parameters. Various metacharacters are defined for special operations; the most interesting is "@," which allows the animator to read in commands stored in a file. We refer to this as calling an *indirect command file*. The indirection facility nests, so any indirect command file can itself use another indirect command file. This makes it possible to define primitive operations and build up more complex ones by collecting them at progressively higher levels. This process could be used to build complex data or define complex motions. It is also the mechanism for restoring a previously saved script or scene definition.

Indirect command files also allow *twixt* to be driven by special purpose programs, which might want explicit control over an object. The special purpose program would write a command file that the animator would tell *twixt* to read in. The process requires only two commands:

! program that generates script on *file*
@ *file*

The first command (a system escape) runs the program that generates the indirect command file; the second command reads it in.

4.4.2 *Abbreviated commands.* The animator does not have to type out complete commands. An underlying set of subroutines finds the minimal substrings required to distinguish one command from another, and the animator can type in any portion of a command that includes its minimal substring.

This feature is implemented by maintaining a table of known commands. At run time (i.e. when *twixt* starts up) it calls a routine that adds commands to this table. As each command is added, it is insertion sorted [11] into the table. A recursive process then examines each command, comparing it with its neighbors to see how many characters they have in common. The resuult is the minimum number of characters necessary for a string comparison to distinguish any command from any other.

The run time binding of this algorithm is important. It allows *twixt* to selectively implement or de-implement commands based on which display device it is using. There are commands that make sense for some pieces of hardware but not others, thus when running on those other devices *twixt* will not recognize those commands.

4.5 *Screen interaction*

We refer to the manipulation of objects as *guiding*, after Zeltzer [20].

4.5.1 *Absolute guiding.* Anything in *twixt* can be manipulated with explicit commands, e.g.,

place *object* at *xyz*.

This mode discourages fine adjustments because it is so tedious to repeatedly type in numbers that are changing by just a small amount. Its advantage is that the animator can explicitly control the object.

4.5.2 *Interactive guiding.* The already mentioned lack of adequate input facilities led to the development of *knob* mode, a name born of irony rather than reality. In this mode, the animator hits the "*x*," "*y*" or "*z*" keys on the terminal repetitively to step a value positively or negatively. Upper case letters have ten times as much effect as lower, and step sizes are under analog control with yet another key. Other keys control what the animator is controlling (e.g., rotation, position, color). The image on the screen changes in real time as each key is struck.

Recently acquired advanced hardware (an Evans & Sutherland Picture System 330) provides more reasonable means for the animators to interact with their images. This system has a tablet, dials and buttons, all of which are sorely needed to facilitate the image design and animation process.

To use these devices, the animator *binds* a control dial to a display parameter, which can be a simple scalar, a scalar component of a vector, a scalar multiplication of a vector, or a scaling control. Simple scalars would be quantities like specular reflection component. The second category would include, for example, the *Y* component of a position. The third category controls blending of vectors and is used for operations like sliding an object along the vector towards the eyepoint. The fourth category allows the animator to apply coarser or finer control to another dial.

4.5.3 *Comment.* The interactive facilities are designed to allow the animator to interact with the image; as such they provide a few general sequences which allow all kinds of manipulation to be performed. The command interface is intended more for administrative details (e.g., show status, or write out script), whereas the interactive manipulation lets the animator construct and edit images.

4.5.4 *Design aids. twixt* provides a frame buffer viewport guide, field guides and a TV aperture guide, any or all which the animator can overlay on the image. The frame buffer guide shows what portion of the 1:1 aspect ratio vector image will appear on a 4:3 aspect ratio frame buffer image. The field guides divide the screen into a number of squares so the animator can do screen coordinate positioning. The TV aperture guide shows which portions of the image will be projected in adequate quality with NTSC video.

5. SUMMARY

We have described the operation of *twixt*, a three-dimensional event driven animation system. We have showed that a careful combination of a terminal oriented command driven interface with graphical input devices results in a system that is device independent, yet powerful enough to perform moderately complex three-dimensional animation in a wide variety of applications. Practically, the friendliness of the system depends on the level of hardware support. The event driven animation approach allows easy extension of4.5.4 the animation capabilities. Without modification to the structure or interface of *twixt*, more sophisticated functions can be introduced to handle interpolation tasks.

Acknowledgements—Frank Crow and Chuck Csuri have constantly encouraged efforts on the development of *twixt*. The Art Education students at the Computer Graphics Research Group have not too loudly put up with the development efforts, especially Susan Van Baerle. John C. Donkin was a big help in getting the photos for this paper. Kevin Reagh did the paste-ups. Frank Crow, Kathy Simpson, and Dave Zeltzer provided a number of cogent comments. The images in this paper were provided by students in the Art Education program at CGRG.

REFERENCES

1. Ronald M. Baecker, Picture-driven animation. In *Interactive Computer Graphics* (Edited by Herbert Freeman). IEEE Computer Society (1980). Originally published in *Conference Proceedings, Spring Joint Computer Conference*. AFIPS, 1969.
2. J. A. Brewer and D. C. Anderson, Visual Interaction with Overhauser Curves and Surfaces. In *Interactive Computer Graphics* (Edited by Herbert Freeman). IEEE Computer Society (1980). Originally published in *Computer Graphics* **11**(2), (1977).
3. N. Burtnyk and M. Wein, Interactive Skeleton Techniques for Enhancing Motion Dynamics in Keyframe Animation. *CACM* **19**(10), (1976).
4. Edwin Catmull, The Problems of Computer-Assisted Animation. *Comput. Graphics* **12**, (1978).
5. Richard Chuang and Glenn Entis, 3-D Shaded Computer Animation—Step-by-Step. *IEEE Comput. Graphics Appl.* **3**, 18–25 (1983).
6. Franklin C. Crow, Shaded Computer Graphics in the Entertainment Industry *Computer* (March 1978). Reprinted in *Tutorial: Computer Graphics*, Kellogg S. Booth, ed., IEEE Computer Society, 1979.
7. Franklin C. Crow, A More Flexible Image Generation Environment. *Comput. Graphics* **16**, 9–18 (1982).
8. Charles Csuri *et al.*, *Towards an Interactive High Visual Complexity Animation System*. Proc. SIGGRAPH 79 (August 1979).
9. Denis Fortin, Jean-Francois Lamy and Daniel Thalmann, A Multiple Track Animator System for Motion Synchronization. *Proc. ACM SIGGRAPH/SIGART Workshop on Motion*, pp. 187–192.
10. Ronald J. Hackathorn, ANIMA II: A 3-D Color Animation System. *Comput. Graphics* **11**, 54–64 (1977).
11. Donald R. Knuth, *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, Reading, MA (1973).
12. Donald R. Knuth, *The Art of Computer Programming: Fundamental Algorithms* (2nd edition). Addison-Wesley, Reading, MA (1975).
13. Nadia Magnenat-Thalmann and Daniel Thalmann, The Use of High-Level 3-D Graphical Types in the Mira Animation System. *IEEE Comput. Graphics Appl.* **3**, 9–16 (1983).
14. William M. Newman and Robert F. Sproull, *Principles of Interactive Computer Graphics*, 2nd edn. McGraw-Hill, New York (1979).
15. Craig W. Reynolds, Computer Animation with Scripts and Actors. *Comput. Graphics* **16**, 289–296 (1982).
16. David F. Rogers and J. Alan Adams, *Mathematical Elements for Computer Graphics*. McGraw-Hill, New York (1976).
17. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, sixth edn. Bell Laboratories (May 1975).
18. Susan L. VanBaerle, *The Uneven Bars*. The Ohio State University (1983). Computer generated film.
19. Lance Williams, BBOP. *Course Notes, Seminar on Three-Dimensional Computer Animation* (July 27, 1982). ACM SIGGRAPH 82.
20. David Zeltzer, Issues in 3-D Computer Character Animation. In *Course Notes, Introduction to Computer Animation*. Minneapolis, MN (July 24, 1982), to be published.
21. D. Zeltzer, Motor Control Techniques for Figure Animation. *IEEE Comput. Graphics Appl.* **2**, 53–59 (1982).

APPENDIX A: EXAMPLE

This typescript implements the example given in section two: A ball jumps up and down while it rotates. It is obviously oversimplified, but a more complex example would have made the paper too long. The animation is built in two steps: First the rotation is designed, then the jumping. Note the animator's use of dials to adjust the ball's position during the second step. The frame constructor inherently composites the two actions with no intervention by the animator. This script works; it is a prettied version of the script that was actually used to do the animation.

```
call /pic/data/ball   # get data instance
                      # first let's work on rotation
rot ball 0 y          # set initial rotation
event 1 ball.ry       # register event
rot ball 720 y        # set final rotation
event 96 ball.ry      # register event
pencil 1 96           # see what animation looks like
slide ball.ry.96 72   # make rotation one second shorter
pencil 1 72           # now see what it looks like
                      # let's work on the jumping
event 30 ball.p       # grab initial position
event 78 ball.p       # which is also final position
place ball at 0 2 0   # apex of jump
dial ball             # adjust
event 54 ball.p       # register apex
pencil 1 78           # look at composited animation
twerp ball.p para     # use parabolic interpolation
pencil                # look at animation again
script jumping        # save script
```

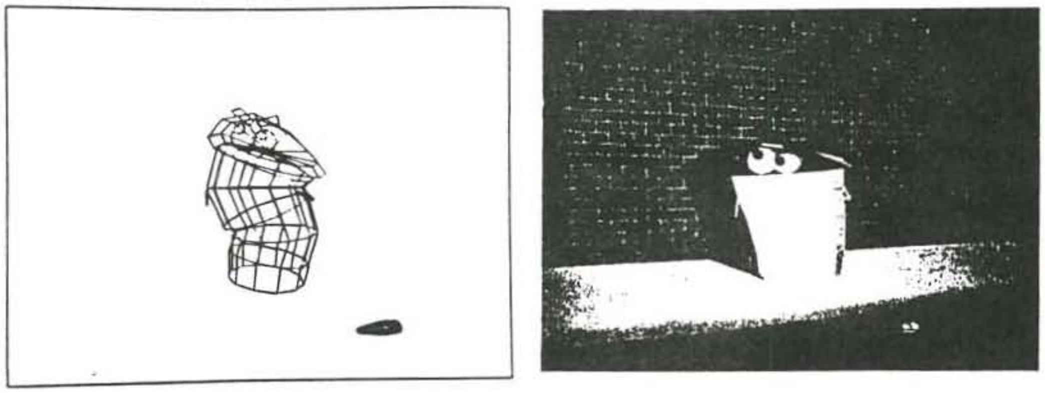


Fig. 1. *Trash* by John Donkin.

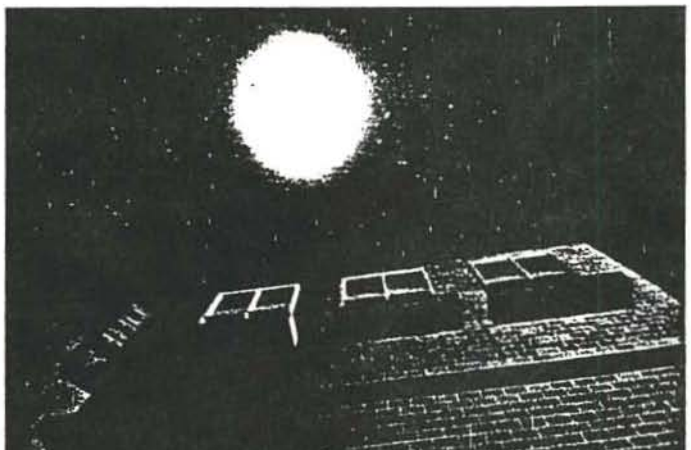


Fig. 2. From *Trash* by John Donkin.

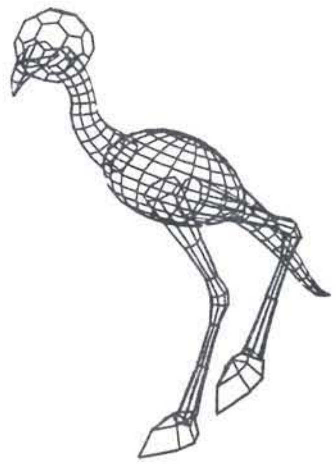


Fig. 3. *Ornithosauraus* by Susan Van Baerle and Doug Kingsbury.

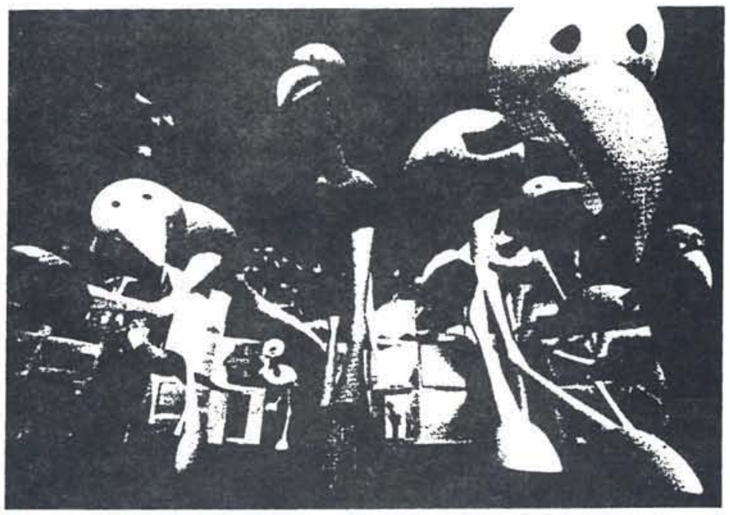


Fig. 4. *Ornithosauri On Parade* by Susan Van Baerle and Doug Kingsbury.

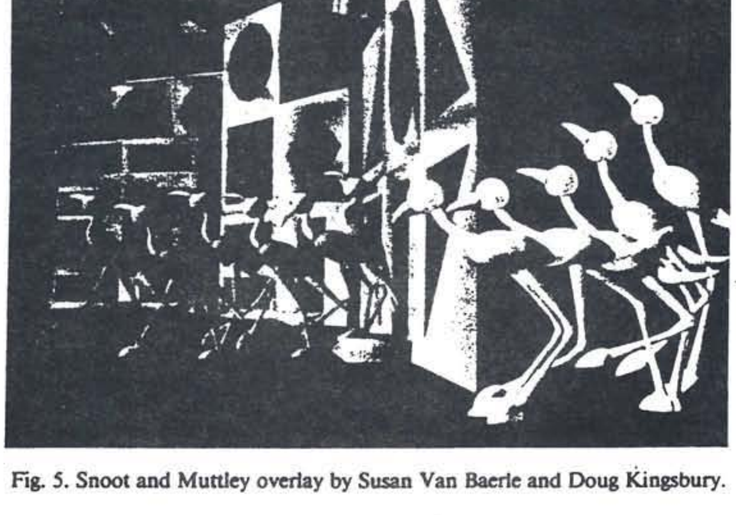


Fig. 5. Snoot and Muttley overlay by Susan Van Baerle and Doug Kingsbury.

Comments on Event Driven Computer Animation

Notes for SIGGRAPH 87 Tutorial
Computer Animation: 3-D Motion Specification and Control

Julian E Gómez
Research Institute for Advanced Computer Science
[{decvax!decwrl}!]julian@riacs.edu

1. Introduction

The most important design goal for an animation system is to not constrain the animator's imagination. The most serious problem with any animation system is the mass of detail required to produce animation.

We don't want a system to force a paradigm on the animator. In particular, it can't require physical laws, although it must be able to supply them when needed. A brief review of classical animation shows this point: although Wily Coyote falls in a fashion that may be related to $d = 1/2at^2$, it usually does not happen until he has been walking on air for a few seconds (the "Cartoon Laws of Motion").

No matter what method is used to describe motion, there is a large amount of data that needs to be specified. A system that provides only one type of movement will not provide the needed flexibility. As Wilhelms [23] points out, with a kinematics description the animator must experiment until the motion looks right, and with a dynamics description the animator must experiment until the desired motion comes out.

The mathematics for computer animation and the techniques for building graphics software have been well explored. Higher level descriptions of animation will be the research area in the future. The last two decades have demonstrated that computer graphics can display animations with adequate form; now it's time to put some effort into constructing animations with content. Recent computer animations [2, 6, 11, 21, 22] show a definite move in this direction. This trend towards *character* animation will tax the capabilities of computer animation systems but produce more interesting animation.

2. Design points for an animation system

A topic related to the design parameters for an animation system is classification of animation systems, a subject treated by Zeltzer [25] and Gomez [9]. Both of these schemes rely on the motion specification mechanism for categorization. Qualifiers are also employed to describe practical aspects of the system, such as playback.

2.1. Interactive

An animation system should be interactive. It's hard to design pictures without looking at a picture of what's being designed and being able to change the picture and see directly what happens.

2.2. Speed

An ideal animation system would draw fancy color pictures in real time. Since this is impractical for the time being, the question becomes one of how much playback can be provided quickly. An acceptable answer is that as soon as the animator has finished adjusting something in the script, he can push a button and have the animation play back in real time. A few seconds delay for precalculation is acceptable; non real-time playback, however, is not. Whereas an animator can find something else to think about for the ten seconds or one minute of precalculation, it's difficult to appreciate motion when it is proceeding at the wrong rate. This is the way things used to be; cel animators wouldn't see any motion until perhaps the next day. In this day and age that's not a valid reason; a valid reason would be something like "this motion needs two minutes of Cray time to evaluate."

The problem is magnified on a multiprogramming system, where in addition to playing the animation at the wrong rate, the system will swap the animation system in and out of the execution queue, causing jerks in the animation.

We will call an animation system that provides acceptable playback an *online* system and designate it as being nice. With current technology, an online system will most likely provide a wireframe display.

2.3. Flexibility

The system shouldn't force the animator to use mechanisms she may not want to use. Sometimes an animator may want a linear spline, even with its attendant lack of continuity in the derivatives. As mentioned in the introduction, sometimes physics may be wanted and sometimes not. The point about flexibility is that the system should not force any motion mechanisms on the animator.

The subjects of splining and splines for computer animation have been discussed adequately in the literature, so these notes won't mention them other than in the description of the *twixt* implementation later.

A useful idea from the *scn_assmblr* system [4] is the ability to substitute module names while interacting with the system. This *loose coupling* makes it easy for the animator to switch data resolution, change lighting algorithm, change anti-aliasing algorithm, change screen resolution, etc. Given this capability for changing parameters at whim and (relatively) immediately obtaining the new result, the animator is given extra ranges of expressive power. When it's trivial to change the way a picture is computed, the user will try those different ways, resulting in effects that may otherwise not have been attempted.

2.4. Extensibility

The basic reason for extensibility is that no matter what facilities the system provides, a need will arise for something else. This is especially true in a research and commercial production environments. Thus the system should include facilities for reconfiguring existing mechanisms or including new ones; it should be *extensible*.

An example of this can be found in the *emacs* editor. It provides a wealth of text operation functions, and the user can write subroutines using these operations to extend the power of the editor. A simple example would be a subroutine that transposes two lines; a more complex one would be an interactive e-mail handling repertoire. Once the user has written or borrowed such a routine, it is as easy to use as a built in *emacs* command, in addition to having the same interface. The point is that the user has the capability of modifying the system to his own desires without rewriting the program.

One way of using this extensibility is to have objects that carry their own behavior with them [13]. Humans, for example, can bend their elbows only so far. It would be nice to include this fact in the "human" abstraction. However, it would also be nice to be able to define a new type, say "human?," that has different or no restrictions on elbow movement.

This notion of dynamic use of the system means the animator can define his own movement criteria and use them in the animation system. This in turn means the animator is effectively reconfiguring the system to his own needs for that animation.

In this context, object oriented programming is a generalization of extensibility. The advantages of object oriented programming extend themselves to any structured system, including one where the constituents are actors and motions rather than lines of source code.

Dynamic components require a rather sophisticated operating system. In particular, a program must be able to load code segments dynamically. Only LISP or Cedar [18] have this notion built into their design. Some efforts have been made towards bringing this attractive capability to UNIX, *e.g.* GEM [14].

2.5. Usability

The system should not be a keyframe system. Originally, a *keyframe* system was one which used *key frames* to control motion. It was designed to facilitate the way hand animation is built [20]. The key frames would be drawn by the animator and the system would interpolate between them. A number of such systems have been implemented [1, 3].

Recently, "keyframe" has been used in a more general sense to mean a system that interpolates between values, whether or not there actually are key frames. These are what Zeltzer calls *guiding* systems [25], indicating that the animator must explicitly describe the animation to be performed. The system will provide splines to smooth out the animator's input.

The use of the terms *guiding* and *key parameter* is strongly preferred over *keyframe,* since the latter term implies there are key frames, in contrast to the first two, which do not. Since contemporary animation systems generally do not work with key frames, this accuracy is desirable.

Finally, the system shouldn't be an extension of a programming language. This forces the animator into a paradigm which has nothing to do with images. It's not necessary (neither is it prohibited) for an animator to know that a *for* loop is necessary to transform the vertices in a database, or that $\cos^n$ is often used in illumination calculations. Furthermore, there is a strong possibility that the detail of dealing with a programming language will distract the animator from the animation.

2.6. Habitability

There are a number of other necessary features in an animation system contributing to its *habitability,* or how nice it is to work in the system. Examples are guarded exit (do not exit unless the script is saved or the user is sure); interactive exception handling (e.g. "File exists - do you want to overwrite it?"); help facilities. Defanti defined many habitability and extensibility requirements in GRASS [5].

2.7. Overall

A point not previously mentioned is that it may require more than one system to perform all these functions, with some sort of hierarchical arrangement between them [25]. This approach would provide different levels of complexity and the corresponding different levels of addressable detail.

3. Event Driven Animation

Event driven animation is an abstraction for describing animation. Rather than describing a specific animation technique, it describes a methodology for describing animation. It is not constrained to describing motion, but it is useful for constructing all aspects of an animation. It can be generalized to any level, thereby providing appropriate degrees of abstraction. A small scale event driven animation system can be implemented easily.

A fundamental concept when dealing with event driven animation is the idea that animation is not limited to moving things around; but also moving the color or the shape or the rate of change of the animation variables. The concept of event driven animation unifies all the different aspects of making an animation. The idea is that all display functions can be treated the same so that operations can be performed on any display function as easily as on any other, freeing the animator from having to use method m_1 to deal with display function $\mathbf{F}_1$ and method m_2 to deal with display function $\mathbf{F}_2$. For example, it's not acceptable for the animator to have to use key joint angles for arm motion and have to use inverse kinematics for leg motion.

Another way of putting this is that animation is not just getting from point A to point B using points C and D to help control a cubic spline; it's dealing with every aspect of making a picture and making the picture move. Thus the mechanisms for performing operations on anything should be similar.

This point can be qualified to a degree. It doesn't make sense to apply vector operations to a scalar value. However, the system should recognize the problem and deal with it, perhaps translating the request to something reasonable. Or the animator could have the options of configuring the system to attempt a translation, ignore the problem, or complain and ask for instructions.

3.1. The Display Function

Consider some arbitrary display function: given some input parameters telling it how to operate, it will take data, process it, and output new values contributing to the picture. Common display functions include the basic geometric transforms such as translation, orientation, and scaling. Other display functions include color, transparency, surface geometry, whether or not to display, joint angles, etc.

It's readily apparent that the datatype required depends on the display function: color is a 3-vector, transparency is usually a scalar, orientation is a 3x3 matrix, surface geometry is a dataset, whether or not to be displayed is a Boolean value, and a methodology for calculating a value is a procedure pointer. When one of these is used to control a display function, we will call it a *control value*.

Mentally we can translate any datatype into a vector of appropriate scalar values. Thus a matrix becomes a 9-vector of real numbers, a dataset becomes a matrix of 3-vectors which is in turn a 3×n vector, a display flag becomes a 1-vector of Boolean values, and a procedure pointer is a pointer valued 1-vector. This point is academic, however, and is mentioned only for formality.

3.2. Definitions

The animation process requires specification of values for every frame of time for every display function implemented in the graphics system. For an arbitrary display function **F** we have a set of control values for it, $\vec{v}_i$. To each of these control value vectors we attach the time at which it is to be used; this construction of the control value and the time we call an *event*. The list of events describing the activity of **F** over the animation we call a ***track***. The track is implicitly sorted in ascending order by event time; sorting should be implemented by the underlying software so the user doesn't have to do it.

Practically, it makes more sense to store events only when an input value for **F** changes, and use a splining technique to generate the inbetween values. Thus interpolation information must also be stored in the event: acceleration/deceleration information, splining method, *etc*. This will be discussed momentarily.

To access values within tracks, we can define an abstract function

$$E(\text{objects},f,t)$$

where *objects* indicates a class of objects, f is the display function, and t is the time. E will return a value appropriate for that display function. The animation controller will have to evaluate the appropriate tracks to calculate that return value vector. The number of events necessary to do this will depend on the display function and the complexity of the splining method, *e.g.* a cubic spline requires four events to work with; a Boolean function requires only the closest preceding event.

Timing in an animation can be changed by changing the frame numbers in events. Track segments can be moved to change the time at which their animation occurs. Track segments can also be multiplied by a factor to expand or compress their length.

3.3. Interpolation

We begin to see a relation between events and curve generation. In fact, the values contained in the events are control points, the frame number is the parameter of interpolation, and the animation for that display function is the result of the generated spline. Here the term *control values* is better then *control points,* to emphasize the fact that event values have arbitrary types, including some which cannot be splined.

There are a plethora of techniques for interpolating or approximating curves. Track animation relies on *patched* curves. Briefly, *patching* refers to the process of "gluing" together splined curves end to end, or surface elements side to side. Continuity in the derivatives across the boundaries, although desirable, is not required. Different interpolation functions can be used to achieve different patches, although the animator can certainly specify a single splining function for the entire duration of the track.

We must assume a track is at least piecewise continuous; otherwise **F** will be undefined at certain frames, a potential source of serious problems. We would also like the first derivative with respect to time, $d\,\mathbf{F}/dt$, or $\dot{\mathbf{F}}$ (velocity) to be continuous; this will prevent sudden jumps in the output values from **F**. If the second derivative $d^2\mathbf{F}/dt^2$, or $\ddot{\mathbf{F}}$ (acceleration) is also continuous, this will prevent sudden jumps in the rate of change of the input values to **F**. If both constraints are met then the output of **F** will be smooth and will change smoothly. See also Smith [17].

In order to calculate a frame in an animation, all tracks are evaluated for the given time. The collection of activity on all tracks inherently generates the animation.

Note that a change in interpolation functions is a change in value, and events can exist just for this purpose. Changes in velocity are also events, *i.e.* specifying values for $d\,\mathbf{F}/dt$ instead of **F** itself.

Keep in mind that interpolation schemes apply to any track, not just position. There is no reason why B-splines can't be used on color or transparency information; for continuity purposes, it's better if they are.

3.4. Generality

The level of abstraction for any track is tied to the intelligence of its *twerper* (interpolator). Position is trivial, rotation matrices are harder, surface geometries are even harder, object collision detection is yet harder, *etc*. The sophistication of the twerper is generally based on the amount of support code available and how much dynamism the operating system can provide.

One way of viewing different levels of tracks is to think of the higher levels compiling down to the lower levels. Just as a high level language is compiled down to a low level language, an abstract track can be compiled down to simpler ones. This allows the animator to deal with varying levels of abstraction, or with animation systems at different levels in Zeltzer's hierarchy. The unifying element among different tracks with different complexities is that the animator has provided certain values at certain points in time to control their behaviors.

Earlier it was mentioned that a control value could be a pointer to a procedure that controls the display function. This procedure would be invoked whenever the animation controller determines that it should be contributing to the calculation of the animation. This is somewhat analogous to "buttons" in Cedar [18, 19], which are modules invoked when a user clicks a button on the screen. In Cedar, part of the process of installing a new button on screen is to tell the window manager what procedure to invoke when the user clicks that button. In the same way, construction of these *inquisitive* events lets the track manager know what procedure to use when evaluation of a track is necessary. This facility is the most powerful aspect of event driven animation, as it allows dynamic control of the animation, where display function controllers can use the current values of other tracks in determining their own values, and thus respond to environmental parameters.

4. twixt

twixt is integrated into the OSU image generation pipeline [24]. This gives the animator a unified environment for dealing with animation and image production.

4.1. Input Methods

As described previously [7], there are a number of ways to describe values to *twixt*. The fancier the display device the user is working, the better these input methods are. Where the input device provides only a limited number of inputs *(i.e.* a bank of control dials), *twixt* provides ways of dynamically changing the assignment of each input device to a control mechanism.

4.2. Layering

The approach to designing animation in *twixt* is *layering,* where the animation is built up in layers of motion. Analogies can be drawn to cel animation, where a frame is built up of a number of cels lying on top of each other. In *twixt*, however, the layers are not pieces of picture, but pieces of motion.

An animator may labor for some time on one particular part of the animation, say the arm of a baseball pitcher throwing a ball. Then he may switch to the ball and work on that. This might be interspersed with quick returns to the arm to perfect some aspect of its motion. It might also be interspersed with work on the snap of the pitcher's head. No commands are required to switch context; the animator is carrying the context in his mind, and the naming scheme in *twixt* allows different ways of specifying the context of an action.

The intent of this approach is that it allows the animator to concentrate on one theme for some time, until he is ready to concentrate on another. It also allows the animator to instantly return to any previous activity in order to modify it. This allows quick implementation of flashes, where the animator remembers or thinks of something that should be done to a sequence already worked on.

4.3. Objects

twixt supports the common practice of constructing object hierarchies, *i.e.* of inserting subtrees into trees to express hierarchical relationships. Thus a scene is actually made of a *forest*[10] of trees. However, the relationships that can be expressed between nodes cover a broader range than that usually available, including operations that cannot be expressed as matrix products. A later section will elaborate.

One nice feature in *twixt* is the way the animator can rapidly switch databases. A pragmatic perusal of animation environments shows that few animations are designed with graphics hardware that can display thousands of vectors in real time. In fact, animators are often working at a station that can handle a few vectors in real time. *twixt* allows the animators to dynamically switch the database used to draw an object. Thus the animator can have rapidly drawn frames of low complexity or slowly drawn frames of high complexity, just by replacing the surface geometry definition of an object. All parameters of an object not related to its geometry are unaltered by this replacement.

On fast graphics hardware this becomes less of a constraint. It will decrease in importance in the near future (see final section of notes).

Objects are named as described in Gomez [7, 8]. Two important points not mentioned in the paper are regular expressions and aliases. Names can contain regular expression characters like the Unix *shell* and *csh;* these characters are handled just as they would be in either of shell. The user can also define alias names, indicating that whenever twixt sees that name, it is to be expanded to all the objects named in the list for that alias. List elements may of course be regular expressions. Furthermore, it is not an error to include an undefined object in an alias list. *twixt* assumes that the animator will bring in that object later, when he is ready for it.

4.4. Track Implementation

twixt implements the following tracks:

position & $d\vec{p}\,/\,dt$
rotation & $d\vec{r}\,/dt$
scale & $d\vec{s}\,/dt$
attach position & $d\vec{a}/\,dt$
color & $d\vec{c}\,/dt$
shininess & $d\overrightarrow{Shininess}\,/dt$
transparency & $d\overrightarrow{Transparency}\,/dt$
surface geometry
display enable flag
attachment
notes

4.4.1. Basic geometrical transformation tracks

Many of these tracks are straightforward: position, scale, color, illumination parameters. Rotation can be treated either as angles around the object's axes or as 3x3 orientation matrices. The former case is easy to implement but non-intuitive, meaning that after a few rotations, it's hard for the animator to make a direct connection between instructing the system to do a rotation and what happens on the screen. This is because the object's axes are themselves transformed, meaning that the rotation is not being applied to the original axes set but the transformed set. In the latter case, matrix interpolation was implemented using a scheme based on a question from my general examinations. This technique has been formalized as quaternion rotations [15, 16].

In addition, there are velocity tracks running alongside each primary track that has a defined derivative (*e.g.* the position track has a derivative but the display enable flag does not). The animator may address any track directly, or its derivative, or both. In the latter case, the velocity track has priority in any conflicts.

As an example, consider an animator who specifies that an object's X position is to be 0 at frame 1 and 20 at frame 24, then specifies that the X velocity is to be 10 units per second. This situation is irreconcilable. The decision to give the velocity track precedence increases the likelihood that the display function will be continuous in its derivatives.

4.4.2. Hierarchy control tracks

The attach position is where a child object is attached to its parent, in terms of the parent's coordinate space. There are various ways of inserting a subtree into a tree:

hang

In this mode, only the offspring's position is transformed by the parent's matrix; the remainder of the matrix is calculated from the child's current display parameters. The parent's scale vector does not propagate down (see *attach* below). This mode is intended for an object that is hanging on another object, such as a rod hanging on a pivot pin. As the pivot pin moves around, the rod must go with it, but it should pivot automatically so it remains in the same orientation.

Implementation is not difficult. To construct the offspring's matrix, first transform it's final position by its parent's matrix and place the result in the bottom row of the matrix. The upper left 3x3 is calculated as usual, with no reference to the parent matrix.

attach

This mode was defined by *scn_assmblr:* parent scale values do not propagate down. It's useful for attaching light sources to other objects, since in the OSU paradigm the scale value of a light source determines its range.

couple

This is a conventional tree builder, where all elements of the parent's matrix propagate to the offspring nodes. This method allows a limited ***squash-and-stretch*** capability.

In actual implementation, *twixt* constructs matrices in a bottom to top fashion. In order to build a frame, each object's matrix must be constructed. To do this, *twixt* goes through its list of objects (which corresponds to visiting each leaf in the forest) and finds which of them has their *newmatrix* flag set, indicating that some display parameter has changed, necessitating recalculation of the matrix. It then travels recursively up the hierarchy tree until it reaches the root of that object subtree, at which point it unwinds, constructing each object's matrix on the way down *iff* that *newmatrix* flag is set and concatenating as appropriate. No matrix is ever computed twice; the *newmatrix* flag is unset to keep that from happening. Thus each node in the tree may be visited more than once, but it won't cause extraneous matrix arithmetic.

There are two ways of removing a subtree from a tree:

detach

Detaches a subtree. The child object (and its children) will no longer be controlled by the parent.

letgo

Detaches a subtree, but maintains the current transformation as a pretransformation for future animation. This is used for objects which are related to another object for part of the animation, then detached to continue own their own way.

An example would be a hand throwing a ball. Initially, the ball would be attached to the hand during the windup. When the ball is released, it is "letgo," so from that point on in time, the hand will have no control over the ball. However, the point at which the ball was let go determines its freeflight, so the transformation at that instant must contribute to the animation following that instant.

The attachment track controls the characteristics of the hierarchy construction. The attach position track is simply a vector showing the attach position. An attachment event simply contains a flag word showing what kind of attach (or detach) is to be performed at what time. If the event is one of the attaches (as opposed to one of the detaches) it also contains a pointer to the new parent.

4.4.3. Surface geometry track

This track controls the surface geometry of an object. Object shapes are interpolated (with flexibilities previously described) between some number of defined geometries. Thus, a *blended* object has no shape it can call it's own; it is defined only when the animation is running. The animator can freeze playback, or play back a single frame, in order to take a look at the current surface geometry. Again, to save memory, the event value field becomes a pointer to another structure that actually defines the characteristics of the actual geometry and contains the data itself.

4.4.4. Notes track

Note events are just that – notes. Animators usually write down all kinds of information on their exposure sheets. Note events are the animator's notes to themselves. When the animation gets to the frame a note event belongs to, the note is printed (the animator can set a flag to enable or disable note printing).

4.5. Track Manipulation

Geometric transformations can be applied to track segments just as they are to objects. Tracks can be scaled, translated, or rotated. These operations are different from changing the frame numbers in events; the former change the values in the events, the latter change the times at which the events occur. Thus the former change the control values themselves; the latter change the timing of the animation.

These track-wise operations are implemented in a simple matter: the animator specifies the track segment by frame numbers, the operation, and the operand. Rotation must be performed on vector or orientation matrix tracks; it does not make sense otherwise.

A track segment copier is provided. This together with the track transformer give the animator *instancing* capability for a track. Just as geometric primitives can be defined and transformed to build more complex objects, tracks can be defined and transformed to eliminate some of the drudge work of animation.

As an example, consider a ball bouncing along a mirror. First the animator animates one bounce of the ball. Then he copies it two or three times, each one shifted by the appropriate time (perhaps two seconds) and the appropriate dislocation. This is the original ball animation. Then the animator makes a second instance of the ball, copies the first one's position track to the second, and multiplies the second ball's Y position track by -1. This is the reflection's animation, and the animator is done. Figure 1 shows value *vs.* time plots for this animation.

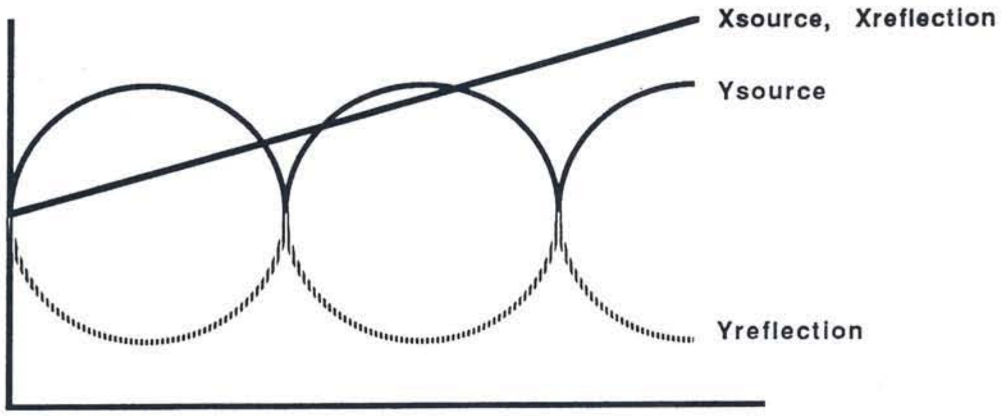


Figure 1.
Plots of ball and reflection positions
(Z is not important for this example)

For a slightly more complex example, suppose the animation was four balls and their reflections bouncing away at right angles from a central point. As before, the animator would animation one ball and its reflection (actually, since this is already done, it's only necessary to read it in from the system). Then this duet would be copied and the copy rotated 90 degrees about the central point. This copy-rotate action is performed twice more, for a total of four balls and their reflections bouncing.

4.6. Record Structures

Following are record structures showing how various entities are implemented. The '☞' character indicates a pointer.

4.6.1. Events

Event structure	
EventTypes	type
...	value
Natural	frame
Natural	easeIn
Natural	easeOut
Twerper	☞ twerper
Logical	dF

Figure 2.
Event record structure

The value field has no type, because it will depend on what the event is being used for. If this structure were being implemented in PASCAL, the event type field would serve as the CASE selector for a variant record.

Whether or not the event is a velocity event can be built into the event type field or separated into its own field as is shown here. The form shown here has some runtime advantages, *e.g.* if some piece of code needs to do something to a color event, whether it's a value or a velocity value, it can work similar to this:

```
if (event.type is Color)
            Doit()
```

instead of like this:

```
if ((event.type is Color) or (event.type is ColorVelocity))
            Doit()
```

Technically, an event structure would be able to handle any kind of display parameter the user desired. Unfortunately, most compilers will simply allocate enough space for the worst case. In the case of a surface geometry definition, it would require a lot of memory. In a global context, most of the memory used would be wasted, since most events are much shorter than surface geometry definitions. Therefore it makes sense to use pointers for events that could take up a lot of memory.

4.6.2. Tracks

Track structure	
Event	☜ events
Twerper	☜ globalTwerper
Event	☜ derivatives
Twerper	☜ globalDTwerper

Figure 3.
Track record structure

The event pointers are *head* pointers, *i.e.* they point to the heads of their respective lists. If a global splining function pointer is non-NULL, then the indicated function should always be used for interpolating that track; otherwise use the patched method as described previously.

An alternative form would be to have a logical flag indicating whether or not to use the global splining function. It's a matter of taste; either way should generate the same number of instructions if the NULL pointer is zero, as it is in C.

4.6.3. Twerpers

Twerper structure	
String	☜ name
...	code

Figure 4.
Interpolating function record structure

The name is used for display purposes, i.e. for telling the user what the name of the function is. It will point to something like "cubic B-spline" or "combination move," *etc.* The other field points to the code implementing that function. It will return whatever's appropriate, typically a floating point blending factor.

4.6.4. Comments

The structures shown here are not the actual declarations used in the program, although they do indicate the information content required. Other fields may be useful for practical purposes. Forward and backward pointers are a help, as doubly linked list traversal is fast. Additional pointers to reduce cross list traversal or avoid indirected lookup also save time. Theoretically, they're not necessary, but faster is better.

Obviously there's more to writing an animation system than what's discussed here. These concepts, however, form the basis around which *twixt* is written. There is a lot more that could be described, but that would be outside the scope of these particular notes. Some additional references along these lines are my dissertation [9] and the user manual [8].

5. Epilogue

An extension to the idea of modifying tracks is to transform them with modifying functions, *i.e.* to filter the display function through time. This would be one way of providing character. After designing a walk cycle, the animator would apply a modifier to provide a particular kind of walk, *e.g.* a limp. There are analogies between this and the NYIT motion postprocessors and Perlin's pixel stream editor [12].

Current developments in fast 3-D raster display systems will not have as much of an impact as advanced user capabilities, because fast hardware is not the hard problem in computer animation. Animators generally desire to see frames of high complexity in full color with advanced surface modeling techniques (note that this is different from actual contemporary situations); advanced 3-D systems generally work only with polygons and simple illumination calculations. The bandwidth required for complex 3-D imagery far exceeds the capability of any current or planned hardware system. Thus the major advances in computer animation will come not from better display units, but from more advanced capabilities available to the animator.

Building an animation system is a nontrivial task. Doing it requires implementations of techniques from all aspects of computer science. It's better to view an animation system as a tool, since its function is to be used, rather than to be an end in itself. Much of the system's success will come from it's users' imagination. But it has to provide them with the appropriate levels of abstraction and the appropriate hierarchy of complexities, where "appropriate" is the nebulous quantity indicating it's not overbearing in normal use but smart enough to help get the job done.

Developers and animators must remain in constant contact over the lifetime of an animation system; otherwise the it will end up being skewed towards the group that built it. The design and development of an animation system should be seen as a symbiotic task between the "technical" types and the "artist" types.

References

1. Baecker, Ronald M, "Picture-driven animation," in *Interactive Computer Graphics*, ed. Herbert Freeman,IEEE Computer Society(1980). Originally published in *Conference Proceedings, Spring Joint Computer Conference*, AFIPS, 1969

2. Bergeron, Daniel and et, al, *Tony di Peltrie*,.ie !"Animation."" Animation. 1985.

3. Catmull, Edwin, "The Problems of Computer-Assisted Animation," *Computer Graphics* **12**(3)(August 1978).

4. Crow, Franklin C, "A More Flexible Image Generation Environment," *Computer Graphics* **16**(3) pp. 9-18 SIGGRAPH-ACM, (July 1982).

5. DeFanti, Thomas A, "The Graphics Symbiosis System -- An Interactive Minicomputer Graphics Language Designed for Habitability and Extensibility," Ph. D. Dissertation, The Ohio State University (March 1973).

6. Donkin, John, *Trash,* Ohio State University CGRG (1984). Animation.

7. Gomez, Julian E, "Twixt: A 3D Animation System," *Computers and Graphics* **9**(9) pp. 291-298 Pergamon Press Ltd., (1985). Reprinted from *Proceedings of Eurographics '84*.

8. Gomez, Julian E, *twixt user manual,* Computer Graphics Research Group, The Ohio State University (1985).

9. Gomez, Julian E, *Computer Display of Time Variant Functions,* The Ohio State University (1985). Ph.D. dissertation

10. Knuth, Donald E., *The Art of Computer Programming Volume 2: Seminumerical Algorithms,* Addison-Wesley Publishing Co., Reading, Mass. (1969).

11. Lasseter, John, *Luxo, Jr.,* Pixar, Inc. (1986). Animation.

12. Perlin, Ken, "An Image Synthesizer," *Computer Graphics* **19**(3)(July 1985).

13. Reynolds, Craig W, "Computer Animation with Scripts and Actors," *Computer Graphics* **16**(3) pp. 289-296 SIGGRAPH-ACM, (July 1982).

14. Schlag, John F., "Eliminating the Dichotomy Between Scripting and Interaction," in *Proc. Graphics Interface '86*, (May 1986).

15. Shoemake, Ken, "Animating Rotation with Quaternion Curves," *Computer Graphics* **19**(3)(July 1985).

16. Shoemake, Ken, "Quaternion Calculus and Fast Animation," in *Tutorial Notes: Computer Animation: 3-D Motion Specification and Control,* ACM SIGGRAPH(July 1987).

17. Smith, Alvy Ray, "Spline Tutorial Notes," in *Tutorial Notes: Computer Animation,* SIGGRAPH(1984).

18. Teitelman, Warren, "The Cedar Programming Environment: A Midterm Report and Examination," CSL-83-11, Xerox PARC (June 1984).

19. Teitelman, Warren, "A Tour Through Cedar," *IEEE Software* **1**(2) pp. 44-73 (April 1984).

20. Thomas, Frank and Johnston, Ollie, *Disney Animation: The Illusion of Life,* Abbeville Press, New York (1981).

21. VanBaerle, Susan and Kingsbury, Doug, *Snoot and Muttly,* Ohio State University CGRG (1984). Animation.

22. Wedge, Chris, *Tuber's Two Step,* Ohio State University CGRG (1985). Animation.

23. Wilhelms, Jane, "Towards Automatic Motion Control," in *Tutorial Notes: Computer Animation: 3-D Motion Specification and Control*, 1986

24. Zeltzer, David, Gomez, Julian, and MacDougal, Paul, "A Tool Set for 3-D Computer Animation," in *Tutorial Notes: Introduction to Computer Animation*, SIGGRAPH(1984).

25. Zeltzer, David, "Toward An Integrated View of 3-D Computer Animation," in *Tutorial Notes: Introduction to Computer Animation*, SIGGRAPH(1984).

Animating Rotation with Quaternion Curves

Ken Shoemake†

The Singer Company
Link Flight Simulation Division

ABSTRACT

Solid bodies roll and tumble through space. In computer animation, so do cameras. The rotations of these objects are best described using a four coordinate system, quaternions, as is shown in this paper. Of all quaternions, those on the unit sphere are most suitable for animation, but the question of how to construct curves on spheres has not been much explored. This paper gives one answer by presenting a new kind of spline curve, created on a sphere, suitable for smoothly in-betweening (i.e. interpolating) sequences of arbitrary rotations. Both theory and experiment show that the motion generated is smooth and natural, without quirks found in earlier methods.

C.R. Classification: G.1.1 [Numerical Analysis] Interpolation—Spline and piecewise polynomial interpolation; G.1.2 [Numerical Analysis] Approximation—Spline and piecewise polynomial approximation; I.2.9 [Artificial Intelligence] Robotics—Manipulators; I.3.5 [Computer Graphics] Computational Geometry and Object Modelling—Curve, surface, solid, and object representation, —Geometric algorithms, languages, and systems, —Hierarchy and geometric transformations

General Terms: Algorithms, Theory

Keywords and phrases: quaternion, rotation, spherical geometry, spline, Bézier curve, B-spline, animation, interpolation, approximation, in-betweening

1. Introduction

Computer animation of three dimensional objects imitates the *key frame* techniques of traditional animation, using key positions in space instead of key drawings. Physics says that the general position of a rigid body can be given by combining a translation with a rotation. Computer animators key such transformations to control both simulated cameras and objects to be rendered. In following such an approach, one is naturally led to ask: What is the best representation for general rotations, and how does one in-between them? Surprisingly little has been published on these topics, and the answers are not trivial.

This paper suggests that the common solution, using three Euler's angles interpolated independently, is not ideal. The more recent (1843) notation of quaternions is proposed instead, along with interpolation on the quaternion unit sphere. Although quaternions are less familiar, conversion to quaternions and generation of in-between frames can be completely automatic, no matter how key frames were originally specified, so users don't need to know—or care—about inner details. The same cannot be said for Euler's angles, which are more difficult to use.

Spherical interpolation itself can be used for purposes besides animating rotations. For example, the set of all possible directions in space forms a sphere, the so-called Gaussian sphere, on which one might want to control the positions of infinitely distant light sources. Modelling features on a globe is another possible application.

It is simple to use and to program the method proposed here. It is more difficult to follow its development. This stems from two causes:. 1) rotations in space are more confusing than one might think, and 2) interpolating on a sphere is trickier than interpolating in, say, a plane. Readers well acquainted with splines and their use in computer animation should have little difficulty, although even they may stumble a bit over quaternions.

2. Describing rotations

2.1 Rigid motion

Imagine hurling a brick towards a plate glass window. As the brick flies closer and closer, a nearby physicist

† Author's current address: 1700 Santa Cruz Ave., Menlo Park, CA 94025

This reprint contains two corrections from the original paper. They are in section 3.2 on page 247 and in the the central element of the matrix M of section I.4 on page 253.

might observe that, while it does not change shape or size, it can tumble freely. Leonhard Euler proved two centuries ago that, however the brick tumbles, each position can be achieved by a single rotation from a reference position. [Euler,1752] [Goldstein] The same is true for any rigid body. (Shattering glass is obviously not a single rigid body.)

While translations are well animated by using vectors, rotation animation can be improved by using the progenitor of vectors, quaternions. Quaternions were discovered by Sir William Rowan Hamilton in October of 1843. The moment is well recorded, for he considered them his most important contribution, the inspired answer to a fifteen-year search for a successor to complex numbers. [Hamilton] By an odd quirk of mathematics, only systems of two, four, or eight components will multiply as Hamilton desired; triples had been his stumbling block.

Soon after quaternions were introduced, Arthur Cayley published a way to describe rotations using the new multiplication. [Cayley] The notation in his paper so closely anticipates matrix notation, which he devised several years later, that it may be taken as a formula for converting a quaternion to a rotation matrix. It turns out that the four values making up a quaternion describe rotation in a natural way: three of them give the coordinates for the axis of rotation, while the fourth is determined by the angle rotated through. [Courant & Hilbert]

Since computer graphics leans heavily on vector operations, it is perhaps easiest to explain quaternions and rotation matrices in terms of these, reversing history. However quaternions can stand on their own as an elegant algebra of space. [Herstein] [Pickert] [MacLane]

2.2 Rotation matrices

That a tumbling brick does not change size, shape, nor "handedness" is mathematically expressed as the preservation of dot products and cross products, since these measure lengths, angles, and handedness. And since the determinant of a 3×3 matrix can be computed as the dot product of one column with the cross product of the other two, determinants are also preserved. Symbolically:

$$\mathrm{Rot}(\mathbf{u}_1)\cdot\mathrm{Rot}(\mathbf{u}_2) = \mathbf{u}_1\cdot\mathbf{u}_2$$

$$\mathrm{Rot}(\mathbf{u}_1)\times\mathrm{Rot}(\mathbf{u}_2) = \mathrm{Rot}(\mathbf{u}_1\times\mathbf{u}_2)$$

$$\det(\mathrm{Rot}(\mathbf{u}_1),\mathrm{Rot}(\mathbf{u}_2),\mathrm{Rot}(\mathbf{u}_3)) = \det(\mathbf{u}_1,\mathbf{u}_2,\mathbf{u}_3)$$

An immediate consequence is that orientation changes must be linear operations, since the preserved operations are; hence they have a matrix representation, M. Using the matrix form of a dot product, $\mathbf{u}_1^t\ \mathbf{u}_2$, we can say more precisely that $(M\ \mathbf{u}_1)^t\ (M\ \mathbf{u}_2) = \mathbf{u}_1^t\ \mathbf{u}_2$, from which it follows that

$$M^t\ M = I\ .$$

That is, the change matrix M is ***orthogonal***; its columns (and rows) are mutually perpendicular unit magnitude vectors. Because M must also preserve determinants, it is a ***special orthogonal*** matrix, satisfying

$$\det(M) = +1\ .$$

It is well known, and anyhow easy to show, that the special orthogonal matrices form a group, $SO(3)$, under multiplication. [MacLane][Goldstein][Misner] In this rotation group, the inverse of M is just M^t, the opposite rotation.

To illustrate, the matrix

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

effects a rotation through an angle of θ around the x axis. After verifying the properties discussed so far, note that the diagonal entries sum to $1+2\cos\theta$. While it is too lengthy to show here, the diagonal sum measures the same quantity for matrices generating rotation around any axis. [MacLane]

2.3 Quaternions

Quaternions, like rotations, also form a non-commutative group under their multiplication, and these two groups are closely related. [Goldstein] [Pickert][Misner] In fact, we can substitute quaternion multiplication for rotation matrix multiplication, and do less computing as a result. [Taylor]

To perform quaternion arithmetic, group the four components into a real part—a scalar, and an imaginary part—a vector. Addition is easy: add scalar to scalar and vector to vector. But our major interest is in multiplication. Start with a simple case: multiply two quaternions without real parts, or more precisely, with zero real parts. The result quaternion has a vector that is the cross product of the two vector parts, and a scalar that is their dot product, negated:

$$\mathbf{u}_1\mathbf{u}_2 = [(-\mathbf{u}_1\cdot\mathbf{u}_2),(\mathbf{u}_1\times\mathbf{u}_2)]\ .$$

It is certainly convenient to encompass both vector products with a single quaternion product. (One early lover of quaternion algebra called vector algebra a "hermaphrodite monster", since it required two kinds of product, each yielding a different type of result.) If one quaternion has only a scalar part, with its vector components all zero, multiplication is just real multiplication and vector scaling. Combining the two effects gives the general rule [Brady]:

$$[s_1,\mathbf{u}_1]\ [s_2,\mathbf{u}_2] = [(s_1s_2-\mathbf{u}_1\cdot\mathbf{u}_2),(s_1\mathbf{u}_2+s_2\mathbf{u}_1+\mathbf{u}_1\times\mathbf{u}_2)]\ .$$

Except for the cross product this looks like complex multiplication, $(a_1+\mathbf{i}b_1)(a_2+\mathbf{i}b_2) = (a_1a_2-b_1b_2) + \mathbf{i}(a_1b_2+a_2b_1)$, as Hamilton intended.†

Quaternions multiply with a cross product because rotations confound axes. To illustrate , place a book in front of you, face up, with the top farthest away. Use this orientation as a reference. Now hold the sides and flip it toward you onto its face, rotating 180 degrees around a left-to-right axis, y. Then, keeping it face down, spin it clockwise 180 degrees around an up-down z axis. Two rotations around two perpendicular axes; yet the total change in orientation must be, according to Euler, a single rotation. Indeed, if you hold the ends of the spine and flip the book 180 degrees around this third, outward-pointing, x axis, you should restore the original orientation. As quaternions, this is —anticipating developments ahead— [0,(0,1,0)] times [0,(0,0,1)] equals [0,(1,0,0)]; the cross product is essential.

Notice how quaternion operations give a new orientation, in "quaternion coordinates", much as translations give a position, relative to some starting reference. A central message of this paper is that quaternion coordinates are best for interpolating orientations. For comparison, imagine using spherical coordinates for translations! Quaternions represent orientation as a single rotation, just as rectangular coordinates represent position as a single vector. Translations combine by adding vectors; rotations, by multiplying quaternions. The separate axes of translations don't interact; the axes of rotations must. Quaternions preserve this interdependence naturally; Euler's angle coordinates ignore it.

2.4 Euler's angles

Why, then, do so many animators use Euler's angles? Mostly, I suspect, because quaternions are unfamiliar. Unlike Euler's angles, quaternions are not taught early in standard math and physics curricula. Certainly there is a plethora of arguments against angle coordinates. Euler's angle coordinates specify orientation as a series of three independent rotations about pre-chosen axes. For example, the orientation of an airplane is sometimes given as "yaw" (or "heading") around a vertical axis, followed by "pitch" around a horizontal axis through the wings, followed by "roll" around the nose-to-tail line. These three angles must be used in exactly the order given because rotations do not commute. The ordering of rotation axes used is a matter of convention, as is the particular set of axes, no matter what the order. For instance some physicists use the body centered axes z-x-z, in contrast to the aeronautics z-y-x. At least a dozen different conventions are possible for which series of axes to use. [Kane][Goldstein] The geometry of orientations in Euler's angle coordinates is contorted, and varies with choice of initial coordinate axes. There is no reasonable way to "multiply" or otherwise combine two rotations. Even converting between rotation matrices and angle coordinates is difficult and expensive, involving arbitrary assumptions and trigonometric functions. In their defense, it must be said that they are handy for solving differential equations—which is how Euler used them. [Euler,1758]

† Hamilton wrote a quaternion as $s+\mathbf{i}v^x+\mathbf{j}v^y+\mathbf{k}v^z$, with $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$. The multiplication rules given before are consequences of this elegant formulation.

3. In-betweening alternatives

3.1 Straight line in-betweening

It is not immediately obvious how to in-between even two rotation keys. What orientations should an object assume on its journey between them? A natural answer is: take the first key as a reference, and represent the second by describing the single rotation that takes you to it, according to Euler's theorem. The in-between orientations should be positioned along that rotation.

If we plot quaternions as points in four-dimensional space, the straight lines between them give orientations interpolating the end points in exactly the above sense. If we plot Euler's angle coordinates instead, the in-between orientations will try to twist around three different axes simultaneously. This angle interpolation treats the three angles of rotation at each key orientation as a three-dimensional vector whose components are interpolated independently from key to key. Paradoxically, we can not rotate simply except around the special axes chosen for composition. We may even encounter so-called "gimbal lock", the loss of one degree of rotational freedom. Gimbal lock results from trying to ignore the cross product interaction of rotations, which can align two of the three axes. Quaternions are safe from gimbal lock, and so have been used for years to handle spacecraft, where it is unacceptable. [Kane][Mitchell]

3.2 How quaternions rotate

Straight lines between quaternions, however, ignore some of the natural geometry of rotation space. If our interpolated points were evenly spaced along a line, the animated rotation would speed up in the middle. To see why, we must look at how a quaternion converts to a rotation matrix. We rotate a vector by a quaternion so: multiply it on the left by the quaternion and on the right by the inverse of the quaternion, treating the vector as $[0,\underline{v}]$.

$$\underline{v}' = \mathrm{Rot}(\underline{v}) = q \quad \underline{v}\; q^{-1}$$

Though it is not obvious, the result will always be a vector, with a zero scalar component. Notice how this guarantees

$$\mathrm{Rot}(\underline{v}_1)\,\mathrm{Rot}(\underline{v}_2) = \mathrm{Rot}(\underline{v}_1\;\underline{v}_2)$$

which implies that dot and cross products are preserved, embedded in the quaternion product.

The inverse of a quaternion is obtained by negating its vector part and dividing both parts by the magnitude squared. For $q = [s, \underline{v}]$,

$$q^{-1} = \frac{1}{||q||^2}[s, -\underline{v}] \; ; \qquad ||q||^2 = s^2 + \underline{v} \cdot \underline{v} \; .$$

Because all effects of magnitude are divided out, any scalar multiple of a quaternion gives the same rotation. (This kind of behavior is not unknown in computer graphics; any scalar multiple of a point in homogeneous coordinates gives the same non-homogeneous point.)

If the scalar part has value w, and the vector part values x, y, and z, the corresponding matrix can be worked out to be

$$M = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}$$

when the magnitude $w^2+x^2+y^2+z^2$ equals 1. The magnitude restriction implies that, plotted in four-dimensional space, these quaternions lie on a sphere of radius one. Deeper investigation shows that such unit quaternions carry the amount of rotation in w, as $\cos \theta/2$, while the vector part points along the rotation axis with magnitude $\sin \theta/2$. The axis of a rotation is that line in space which remains unmoved; but notice that's exactly what happens when scalar multiples of $\underline{v}$ are rotated by $[s, \underline{v}]$. Because the cross product drops out, multiplication commutes, q^{-1} meets q, mutual annihilation occurs, and the vector emerges unscathed. Summing the matrix diagonal leads to the formula stated for w. The sum equals $4w^2-1$, but must also be $1+2\cos\theta$. A trig identity, $\cos 2\theta = 2\cos^2\theta - 1$, finishes the demonstation.

3.3 Great arc in-betweening

This sphere of unit quaternions forms a sub-group, S^3, of the quaternion group. Furthermore, the spherical metric of S^3 is the same as the angular metric of $SO(3)$. [Misner] From this it follows that we can rotate without speeding up by interpolating on the sphere. Simply plot the two given orientations on the sphere and draw the great circle arc between them. That arc is the curve where the sphere intersects a plane through the two points and the origin. We sped up before because we were cutting across instead of following the arc; otherwise the paths of rotation are the same.

A formula for *s*pherical *l*inear int*erp*olation from q_1 to q_2, with parameter u moving from 0 to 1, can be obtained two different ways. From the group structure we find

$$\text{Slerp}(q_1, q_2; u) = q_1(q_1^{-1}q_2)^u \; ;$$

while from the 4-D geometry comes†

$$\text{Slerp}(q_1, q_2; u) = \frac{\sin (1-u)\theta}{\sin \theta} q_1 + \frac{\sin u\theta}{\sin \theta} q_2 \, ,$$

where $q_1 \cdot q_2 = \cos \theta$. The first is simpler for analysis, while the second is more practical for applications.

But animations typically have more than two key poses to connect, and here even our spherical elaboration of simple linear interpolation shows flaws. While orientation changes seamlessly, the direction of rotation changes abruptly. In mathematical terms, we want higher order continuity. There are lots of ways to achieve it—off the sphere; unfortunately we've learned too much.

3.4 Rotation geometry and topology

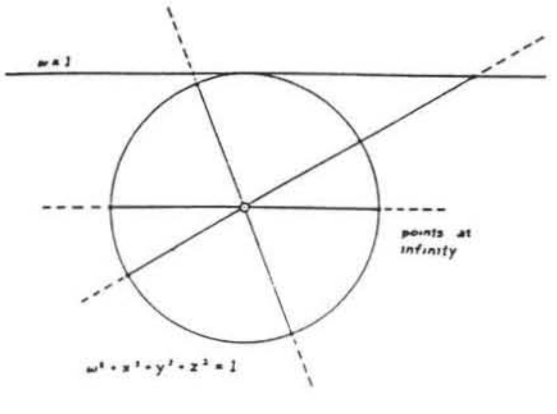
No matter what we do in general quaternion space, the ultimate effect must be interpreted via the sphere; so we had best work there in spite of the difficulty. It is important to grasp this point. The metric structure, hence the intrinsic geometry, of the rotation group $SO(3)$ is that of a sphere. Over small regions, meaning in this case small rotation angles, a sphere looks as if it is flat. But if we go far enough along a "straight line", we end up back where we started. What could be more evident about rotations? Their very essence is moving in circles. Looking back to the book-turning experiment, the confounding of axes is like traveling on a sphere: if we go in some direction to a quarter of the way around the sphere, turn 90 degrees, travel the same distance, then turn and travel again, we will arrive back home, coming in at right angles to the direction we headed out. Even more revealing, we can leave the north pole in any direction and end up at the south pole, just as we can rotate 360 degrees around any axis and end up oriented the same way.

Local geometry does not, however, determine global topology. Contradictory though it may seem, the geometry curves like a sphere, but the topology says north and south poles are the same! In fact, each pair of opposite points represents the same rotation. The reader may preserve sanity through two expedients. One is to see that this, like homogeneous coordinates, is geometry under perspective projection. The second is to restore spherical topology · by including "entanglements". Physically, taking an object with strings attached and rotating it 360 degrees leaves the strings tangled; yet—most odd—rotating 720 degrees does not. [Misner][Gardner]

Accepting the topological oddity is more useful here, but it leaves a minor inconvenience. Namely, when converting an orientation in some foreign form, such as a matrix, to a quaternion form, which quaternion should we choose? Which side of the sphere? An answer that works well is this. Construct a string of quaternions through which to interpolate by choosing

† Glenn Davis suggested this formula.

each added quaternion on the side closest to the one before. Then small changes in orientation will yield small displacements on the sphere.



Representing a projective plane

3.5 Splines

We are left with the problem of constructing smooth curves on spheres. About a hundred years after quaternions appeared, Isaac Schoenberg published a two part attack on ballistics and actuarial problems, using what he called splines. [Schoenberg] Named by analogy to a draftman's tool, these are interpolating curves constructed from cubic polynomial pieces, with second order continuity between pieces. Cubic splines solve an integral equation which says to minimize the total "wiggle" of the curve, as measured by the second derivative. These interpolants are very popular, and the equation can be augmented with Lagrange multipliers to constrain the solution curves to lie on a sphere [Courant & Hilbert]; yet there are problems. First, the augmented equation is much more difficult and expensive to solve. Second, the curve must adjust everywhere if one of the points changes; that is, we have no local control.

3.6 Bézier curves

While Schoenberg invented splines based on numerical analysis, Pierre Bézier invented a class of curves, now called by his name, based on geometrical ideas. In fact, he showed how to find points on such a curve by drawing lines and splitting them in regular proportions. [Bézier] This is exactly what is needed. We already know how to do the equivalent—draw great arcs and proportions of arcs—on a sphere. A complete solution needs only a little more.

4. Spherical Bézier curves

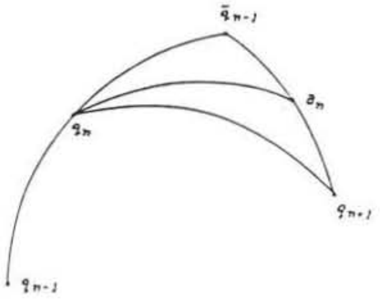
4.1 Joining curves

Bézier curves go through only their first and last defining points, but we want to interpolate all our orientations. The trick is to splice together short Bézier curves in the manner of splines. Their creator showed an easy way to do this which guarantees first order continuity, probably enough for us. As the curve goes through its end points it is tangent to its end segments. Line up the segments across a join, match their lengths, and the curves will piece together smoothly. If the key orientations are placed at joints, then each short curve moves us from one key to the next, because each piece passes through its ends.

Now, although the two segments abutting a curve junction should match each other, one of the segments can be chosen freely. These choices determine the axis and speed of rotation as we pass through the keys. The burden of choice can be passed to the animator of course, but automation is feasible, and generally preferable.

4.2 Choosing joint segments

Spherical linear interpolation gives two conflicting arc segments at a joint, one on each side. Smooth the difference with an even compromise, aiming for a point halfway between where the incoming segment would proceed, and where the outgoing segment must arrive.†



Constructing a point for tangent

Given successive key quaternions q_{n-1}, q_n, q_{n+1} interpretted as 4-D unit vectors, the computation for a segment point a_n after q_n is

$$a_n = \text{Bisect}(\text{Double}(q_{n-1}, q_n), q_{n+1}) \text{ ,}$$

where

$$\text{Double}(p,q) = 2(p \cdot q)q - p \text{ ;}$$

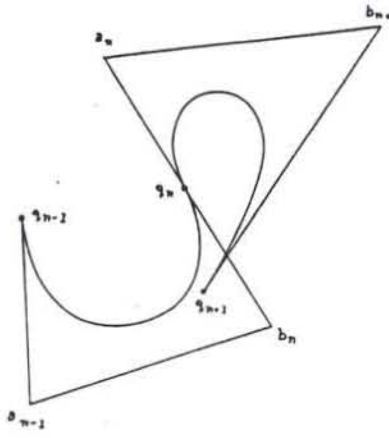
$$\text{Bisect}(p,q) = \frac{p+q}{||p+q||} \text{ .}$$

The matching point for the segment before q_n should be

$$b_n = \text{Double}(a_n, q_n)$$

† For the numerically knowledgeable, this construction approximates the derivative at points of a sampled function by averaging the central differences of the sample sequence. [Dahlquist & Björk]

to ensure a smooth join, regardless of how a_n is chosen.



Splicing Bèzier segments together

4.3 Evaluating on the sphere

Everything is now in hand to imitate Bézier's curve technique. Each short curve is defined by four quaternions, q_n, a_n, b_{n+1}, q_{n+1}. Let the parameter u vary from 0 to 1 as the curve departs q_n towards a_n and arrives at q_{n+1} tangent to the arc from b_{n+1}. Spherically interpolate by proportion u between q_n and a_n, a_n and b_{n+1}, b_{n+1} and q_{n+1}, to obtain three new quaternions. Then interpolate between those to get two more; and finally interpolate again, reducing to a single point. Abbreviating Slerp$(p,q;u)$ as $(p:q)_u$, the computation looks like this:

$$q_n = p_0^{(0)}$$

$$(p_0^{(0)}:p_1^{(0)})_u = p_0^{(1)}$$

$$a_n = p_1^{(0)} \qquad (p_0^{(1)}:p_1^{(1)})_u = p_0^{(2)}$$

$$(p_1^{(0)}:p_2^{(0)})_u = p_1^{(1)} \qquad (p_0^{(2)}:p_1^{(2)})_u = p_0^{(3)} = q_{n+u}$$

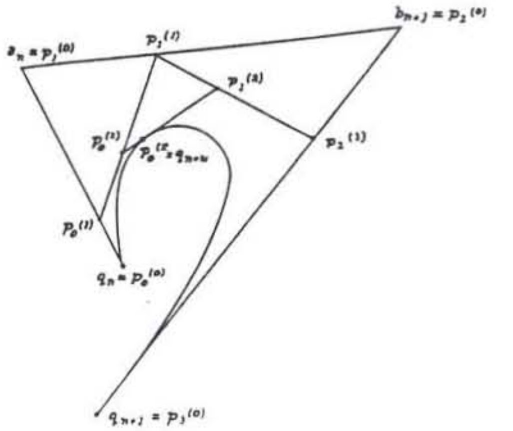
$$b_{n+1} = p_2^{(0)} \qquad (p_1^{(1)}:p_2^{(1)})_u = p_1^{(2)}$$

$$(p_2^{(0)}:p_3^{(0)})_u = p_2^{(1)}$$

$$q_{n+1} = p_3^{(0)}$$

4.4 Tangents revisited

A simple check proves the curve touches q_n and q_{n+1} at its ends. A rather challenging differentiation shows it is tangent there to the segments determined by a_n and b_{n+1}. However, as with Bézier's original curve, the magnitude of the tangent is three times that of the segment itself. That is, we are spinning three times faster than spherical interpolation along the arc. Fortunately we can correct the speed by merely truncating the end segments to one third their original length, so that a_n is closer to q_n and b_{n+1} closer to q_{n+1}.



Calculating a Bèzier curve point recursively

5. Results

5.1 The grand scheme

What have we ended up with? An animator sits at a workstation and interactively establishes a sequence of keys for, say, camera orientation. The interpolating algorithm does not depend on the nature of the interface the animator sees; all needed information is contained in the sequence of keys. Probably the orientations will be represented internally as matrices, so a conversion step follows. The matrices are "lifted" to a sequence of neighboring quaternions, q_n, on the unit sphere. Each quaternion within the sequence will become the endpoint of two spherical Bézier curves. Between each quaternion pair, q_n and q_{n+1}, two additional points, a_n and b_{n+1}, are added to control motion through the joints. At this point, time becomes a parameter along the composite curve. As the frame number increments, the parameter enters and leaves successive curve pieces. Within each piece a local version of the parameter is adjusted to run from 0 to 1. Now the Bézier geometric construction comes into play, producing an interpolated quaternion, q_{n+u}, from q_n, a_n, b_{n+1}, q_{n+1}, and the local parameter, u. Finally the mint-fresh interpolated quaternion is transmuted into a matrix, to be used in rotating a list of object vectors for rendering.

5.2 Properties

A look at one special case is revealing. Suppose all the points to interpolate are spread along a single arc. This means they represent different amounts of rotation around a single axis, in which case quaternion multiplication commutes. Under these special conditions, the formula for the curve sections reduces to

$$q_{n+u} = q_n^{(1-u)^3} \, a_n^{3(1-u)^2u} \, b_{n+1}^{3(1-u)u^2} \, q_{n+1}^{u^3}$$

When this is compared to the standard Bézier polynomial, $p_n(1-u)^3 + a_n 3(1-u)^2 u + b_{n+1} 3(1-u)u^2 + q_{n+1}u^3$, it is apparent that addition and multiplication have become multiplication and exponentiation. Of course, when the points are not on one arc, commutativity fails, so the formula looks much messier.

In the interesting restricted case when the points are spaced evenly and consecutively around an arc, the resulting animation behaves exactly as we would hope: we get smooth, constant speed rotation around the appropriate axis. Notice that we can choose *any* axis for this rotation. This is clearly preferable to interpolation with Euler's angles, where the coordinate axes are special. A more subtle property of *all* quaternion interpolation is that the motion is independent of coordinate axes. So, for example, if we design a move, then rotate the coordinate system arbitrarily, the geometry of the motion will not change. Euler interpolants, unfortunately, will do wildly different things.

5.3 Applicability

Rotations in space are significantly more complicated than rotations in a plane. It is easy to deal with the latter, since only one parameter is involved. Quaternions are out of place in a plane. Joint control in robotics simulations has its own highly specialized body of techniques; and though quaternions have shown up in the literature, they seem less useful in that context. [Brady][Taylor] However, B.K.P. Horn has used a tessellation of the quaternion unit sphere to identify the orientation of an object from its extended Gaussian image; a good reference is [Brou]. Non-rigid motion obviously needs to be handled specially. But for moving a camera eye-point, and for many kinds of object motion, quaternion interpolation has strong advantages.

5.4 Comparisons and complaints

Cost advantages are difficult to estimate. Converting a matrix to a quaternion requires only one square root and three divides plus some adds, at worst. Converting back requires 9 multiplies and 15 adds. While the conversions don't use trigonometric functions, the arc proportioning does. For comparison, angle interpolation requires several trigonometric functions as well as quite a few multiplies and adds to create each interpolated matrix. My experience is that the Bézier scheme is comfortably fast enough for design work, which is the only time speed has mattered. (If, for some application, more speed is essential, non-spherical quaternion splines will undoubtedly be faster than angle interpolation, while still free of axis bias and gimbal lock.)

These interpolants are not perfect, of course. Like all interpolants, they can develop kinks between the interpolated points. There are simple algorithms for adding new sequence points to ordinary splines without altering the original curve [Boehm]; they do not work for this interpolant. And if these curves can be shown to satisfy some variational principal, it will be by chance. It is useful to do this, because any solution to an integral equation like that for splines admits subdivision [Lane et al]; minimum curvature between end points implies minimum curvature between intermediate points as well. Along these lines, Gabriel and Kajiya, motivated by quaternions, have been developing a technique to find splines on arbitrary Reimannian manifolds by solving differential equations. [Gabriel & Kajiya]

6. Questions

Future research could answer some interesting practical questions. What are these spherical Bézier curves? Is there some abstract characterization of them? Or is there some related interpolant that is well-characterized? In light of the success of the geometric adaptation approach, it appears reasonable to apply the idea to B-splines, which also have a known geometric evaluation technique. [Gordon & Riesenfeld] How do spherical B-splines behave? Is it possible to add new points to a sequence for either kind of curve without disturbing it? How? Can B-splines be made to interpolate, not just approximate, with a simple adjustment of control points? Is there a way to construct a curve parameterized by arc length? This would be very useful. What is the best way to allow varying intervals between sequence points in parameter space? Abandoning the unit sphere, one could work with the four-dimensional Euclidean space of arbitrary quaternions. How do standard interpolation methods applied there behave when mapped back to matrices? Note that we now have little guidance in picking the inverse image for a matrix, and that cusp-free $\mathbf{R}^4$ paths do not always project to cusp-free S^3 paths.

However these questions are answered, quaternion spline interpolants already offer a well-behaved improvement over traditional techniques. They are simple to use, simple to implement, robust, efficient, consistent, and flexible. More research would make them even more so.

7. Acknowledgments

This work was begun for an animation system I designed and implemented at Singer-Link. Several people there deserve thanks, but I especially thank Glenn Davis, who befriended me with his good humor and mathematical efforts as I struggled through trying times.

I prefer not to invent the wheel if I can find the plans; so I pestered Don Venhaus, Brian Barsky, Tom Duff, Lance Williams, and Jim Blinn, whom I thank for their time, their comments, and their assurances that they had not seen this particular wheel roll past PDI, Berkeley, Lucasfilm, NYIT, or JPL.

Thanks to everyone at Pacific Data Images for the interest and encouragement that got me started.

The folks at Ridge Computer were generous above and beyond the call of customer support in letting me use their Imagen typesetting system to produce this paper.

Lastly, I thank Nori Hall for commenting on numerous drafts, and more.

References

1. BÉZIER, P.E., *Numerical Control — Mathematics and Applications,* John Wiley and Sons, London (1972).

2. BOEHM, WOLFGANG, "Inserting new knots into B-spline curves," *Computer-Aided Design* **12**(4) pp. 199-201 (July 1980).

3. BRADY, MICHAEL, "Trajectory Planning," in *Robot Motion: Planning and Control,* ed. Michael Brady, John M. Hollerbach, Timothy L. Hohnson, Tomas Lozano-Perez and Matthew T. Mason,The MIT Press (1982).

4. BROU, PHILIPPE, "Using the Gaussian Image to Find the Orientation of Objects," *The International Journal of Robotics Research* 3(4) pp. 89-125 (Winter 1984).

5. CAYLEY, ARTHUR, "On certain results relating to quaternions," *Philosophical Magazine* **xxvi** pp. 141-145 (February 1845).

6. COURANT, R. AND HILBERT, D., *Methods of Mathematical Physics, Volume I,* Interscience Publishers, Inc., New York (1953).

7. DAHLQUIST, GERMUND AND BJÖRCK, ÅKE, *Numerical Methods,* Prentice-Hall, Inc., Englewood Cliffs, N.J. (1974). Translated by Ned Anderson.

8. EULER, LEONHARD, "Decouverte d'un nouveau principe de mécanique (1752)," pp. 81-108 in *Opera omnia, Ser. secunda, v. 5,* Orell Füsli Turici, Lausannae (1957).

9. EULER, LEONHARD, "Du mouvement de rotation des corps solides autour d'un axe variable (1758)," in *Opera omnia, Ser. secunda, v. 8,* Orell Füsli Turici, Lausannae ().

10. GABRIEL, STEVEN A. AND KAJIYA, JAMES T., "Spline Interpolation in Curved Manifolds," , (1985). Submitted

11. GARDNER, MARTIN, *New Mathematical Diversions from Scientific American,* Fireside, St. Louis, Missouri (1971). Chapter 2

12. GOLDSTEIN, HERBERT, *Classical Mechanics, second edition,* Addison-Wesley Publishing Company, Inc., Reading, Mass. (1980). Chapter 4 and Appendix B.

13. GORDON, WILLIAM J. AND RIESENFELD, RICHARD F., "Bernstein-Bézier methods for the computer-aided design of free-form curves and surfaces," *J. ACM* **21**(2) pp. 293-310 (April 1974).

14. GORDON, WILLIAM J. AND RIESENFELD, RICHARD F., "B-spline curves and surfaces," in *Computer Aided Geometric Design,* ed. Robert E. Barnhill and Richard F. Riesenfeld,Academic Press, New York (1974).

15. HAMILTON, SIR WILLIAM ROWAN, "On quaternions; or on a new system of imaginaries in algebra," *Philosophical Magazine* **xxv** pp. 10-13 (July 1844).

16. HERSTEIN, I.N., *Topics in Algebra, second edition,* John Wiley and Sons, Inc., New York (1975).

17. KANE, THOMAS R., LIKINS, PETER W. AND LEVINSON, DAVID A., *Spacecraft Dynamics,* McGraw-Hill, Inc. (1983).

18. LANE, JEFFREY M., CARPENTER, LOREN C., WHITTED, TURNER, AND BLINN, JAMES F., "Scan line methods for displaying parametrically defined surfaces," *Comm. ACM* **23**(1) pp. 23-34 (January 1980).

19. MACLANE, SAUNDERS AND BIRKHOFF, GARRETT, *Algebra, second edition,* Macmillan Publishing Co., Inc., New York (1979).

20. MISNER, CHARLES W., THORNE, KIP S., AND WHEELER, JOHN ARCHIBALD, *Gravitation,* W.H. Freeman and Company, San Francisco (1973). Chapter 41 — Spinors.

21. MITCHELL, E.E.L. AND ROGERS, A.E., "Quaternion Parameters in the Simulation of a Spinning Rigid Body," in *Simulation The Dynamic Modeling of Ideas and Systems with Computers,* ed. John McLeod, P.E., (1968).

22. NEWMAN, WILLIAM M. AND SPROULL, ROBERT F., *Principles of Interactive Computer Graphics, second edition,* McGraw-Hill, Inc., New York (1979). Chapter 21 — Curves and surfaces.

23. PICKERT, G. AND STEINER, H.-G., "Chapter 8 — Complex numbers and quaternions," in *Fundamentals of Mathematics, Volume I — Foundations of Mathematics: The Real Number System and Algebra,* ed. H. Behnke, F. Bachmann, K. Fladt, and W. Süss, (1983). Translated by S.H. Gould.

24. SCHMEIDLER, W. AND DREETZ, W., "Chapter 11 — Functional analysis," in *Fundamentals of Mathematics, Volume III — Analysis,* ed. H. Behnke, F. Bachmann, K. Fladt, and W. Süss,MIT Press, Cambridge, Mass. (1983). Translated by S.H. Gould.

25. SCHOENBERG, I.J., "Contributions to the problem of approximation of equidistant data by analytic functions," *Quart. Appl. Math.* **4** pp. 45-99 and 112-141 (1946).

26. SMITH, ALVY RAY, "Spline tutorial notes," Technical Memo No. 77, Computer Graphics Project, Lucasfilm Ltd. (May 1983).

27. SÜSS, W., GERICKE, H., AND BERGER, K.H., "Chapter 14 — Differential geometry of curves and surfaces," in *Fundamentals of Mathematics, Volume II — Geometry,* ed. H. Behnke, F. Bachmann, K. Fladt, and W. Süss,MIT Press (1983). Translated by S.H. Gould.

28. TAYLOR, RUSSELL H., "Planning and Execution of Straight Line Manipulator Trajectories," in *Robot Motion: Planning and Control,* ed. Michael Brady, John M. Hollerbach, Timothy L. Hohnson, Tomas Lozano-Perez and Matthew T. Mason,The MIT Press (1982).

Appendix I—Conversions

I.1 Quaternion to matrix

Using the restriction that $w^2+x^2+y^2+z^2 = 1$ for a quaternion $q = [w,(x,y,z)]$, the formula for the corresponding matrix is

$$M = \begin{pmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{pmatrix}.$$

If the quaternion does not have unit magnitude, an additional 4 multiplies and divides, 3 adds, and a square root will normalize it. (For the matrix conversion, the square root can be avoided in favor of divides if desirable.) Now we can obtain the operation count for creating the matrix. Most terms of the entries are a product of two factors, one of which is doubled. So we proceed as follows. First double x, y, and z, and form their products with w, x, y, and z. That will take 3 adds and 9 multiplies. Then form the sum for each of the 9 entries using 1 add each, plus an extra add for each of the 3 diagonal elements, for a total of 12 adds. Thus 9 multiplies and 15 adds suffice to convert a unit quaternion to a matrix.

I.2 Matrix to quaternion

An efficient way to determine quaternion components w, x, y, z from a matrix is to use linear combinations of the entries M_{mn}. Notice that the diagonal entries are formed from the squares of the quaternion components, while off-diagonal entries are the sum of a symmetric and a skew-symmetric part. Thus linear combinations of the diagonal entries will isolate squares of components; sums and differences of opposite off-diagonal entries will isolate products among x, y, and z and products with w. Using off-diagonals risks dividing by a component that may be zero, or within ϵ (the machine precision) of zero. However we can avoid that pitfall, and easily compute all components as follows.

```
$w^2 = 1/4\,(1 + M_{11} + M_{22} + M_{33})$
$w^2 > \epsilon$ ?
  TRUE
    $w = \sqrt{w^2}$
    $x = (M_{23} - M_{32})\,/\,4w$
    $y = (M_{31} - M_{13})\,/\,4w$
    $z = (M_{12} - M_{21})\,/\,4w$
  FALSE
    $w = 0$
    $x^2 = -1/2\,(M_{22} + M_{33})$
    $x^2 > \epsilon$ ?
      TRUE
        $x = \sqrt{x^2}$
        $y = M_{12}\,/\,2x$
        $z = M_{13}\,/\,2x$
      FALSE
        $x = 0$
        $y^2 = 1/2\,(1 - M_{33})$
        $y^2 > \epsilon$ ?
          TRUE
            $y = \sqrt{y^2}$
            $z = M_{23}\,/\,2y$
          FALSE
            $y = 0$
            $z = 1$
```

No more than one square root, three divides, and a few adds and binary scales are required for any conversion.

I.3 Euler angles to quaternion

There are twelve possible axis conventions for Euler angles. The one used here is ***roll***, ***pitch***, and ***yaw***, as used in aeronautics. A general rotation is obtained by first yawing around the **z** axis by an angle of ϕ, then pitching around the **y** axis by θ, and finally rolling around the **x** axis by ψ. Using the way quaternion components describe a rotation, we first obtain a quaternion for each simple rotation.

$$q_{roll} = [\cos\frac{\psi}{2},(\sin\frac{\psi}{2},0,0)]$$

$$q_{pitch} = [\cos\frac{\theta}{2},(0,\sin\frac{\theta}{2},0)]$$

$$q_{yaw} = [\cos\frac{\phi}{2},(0,0,\sin\frac{\phi}{2})]$$

Multiplying these together in the right order gives the desired quaternion $q = q_{yaw}\, q_{pitch}\, q_{roll}$, with components

$$w = \cos\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2}$$

$$x = \sin\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} - \cos\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2}$$

$$y = \cos\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2}$$

$$z = \cos\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2} - \sin\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2}$$

I.4 Euler angles to matrix

Combining the results of the previous two conversions gives

$$M = \begin{pmatrix} \cos\theta\cos\phi & \cos\theta\sin\phi & -\sin\theta \\ \sin\psi\sin\theta\cos\phi-\cos\psi\sin\phi & \sin\psi\sin\theta\sin\phi+\cos\psi\cos\phi & \cos\theta\sin\psi \\ \cos\psi\sin\theta\cos\phi+\sin\psi\sin\phi & \cos\psi\sin\theta\sin\phi-\sin\psi\cos\phi & \cos\theta\cos\psi \end{pmatrix}.$$

where ψ, θ, and ϕ are the angles of roll, pitch, and yaw, respectively.

I.5 Matrix to Euler angles

While converting a matrix to a unit quaternion only involves the sign ambiguity of square roots, converting to Euler angles involves inverse trigonometric functions, as we can only directly determine the sin's and cos's of the angles. Some convention, such as principle angles, must be adopted. However interpolation paths will vary greatly, depending on choice of angles. Setting that problem aside, here's a way to extract the sin's and cos's. Looking at the previous equation, $\sin\theta$ can be read off directly as $-M_{13}$. Use the trigonometric identity $\cos\theta = \pm\sqrt{1-\sin^2\theta}$ to compute $\cos\theta$ to within a sign, which is the best we can do. Assuming $\cos\theta$ is not zero, obtain the sin's and cos's of the other angles from

$$\sin\theta = -M_{13}$$

$$\cos\theta = \sqrt{1-\sin^2\theta}$$

$$\sin\psi = M_{23} / \cos\theta$$

$$\cos\psi = M_{33} / \cos\theta$$

$$\sin\phi = M_{12} / \cos\theta$$

$$\cos\phi = M_{11} / \cos\theta$$

If $\cos\theta$ is zero, then we must avoid dividing by zero. It also becomes impossible to distinguish roll from yaw. Adopting the convention that the yaw angle ϕ is 0 allows

$$\sin\psi = -M_{32}$$

$$\cos\psi = M_{22}$$

$$\sin\phi = 0$$

$$\cos\phi = 1$$

From these values a two argument $\tan^{-1}$ will give angles between $-\pi$ and $+\pi$, or 0 and 2π, or some other conventional range; take your pick. (For a faster conversion, just compute, say, $\sin^{-1}$ and check the sign of the cosine term with respect to $\cos\theta$.) Because of the uncertainties of square roots, inverse trigonometric functions, and yaw-roll separation, matrix to Euler angle conversion is inherently *very* ill-defined.

I.6 Quaternion to Euler angles

Use the most straight-forward approach: convert the quaternion to a matrix, then the matrix to Euler angles. Of course it is unnecessary to compute matrix elements that are never used. This conversion is also unavoidably ill-defined, as quaternions contain no more information about angles than matrices do.

Quaternion Calculus and Fast Animation

Ken Shoemake

1700 Santa Cruz Avenue
Menlo Park, California 94025

Abstract

Like aerospace engineers, robotics researchers, and James Clerk Maxwell before them, graphics programmers can benefit from quaternion calculus. Quaternions of unit magnitude give a four-component system of coordinates covering all orientations in space without singularities. No system of three components can do this, and more components add redundancy without advantage. Animation brings new demands: Construct smooth spline paths through rotation space, and do it efficiently. Rotation space is curved, making true splines costly. The quaternion unit hypersphere has the correct curvature; thus "straight lines" are great arcs, and "linear" interpolation is no longer dirt cheap. Smoother curves split several arcs per point. This paper presents a new algorithm for C^1 interpolation that splits only three arcs per point, the minimum necessary for tangent continuity. Using other methods described here, even more speed is possible. These methods include fast arc interpolation, fast quaternion multiplication, and fast conversion between quaternions and matrices. On the way to achieving these results, quaternion calculus is explained.

C.R. Classification: G.1.1 [Numerical Analysis] Interpolation — Spline and piecewise polynomial interpolation; G.1.2 [Numerical Analysis] Approximation — Spline and piecewise polynomial approximation; I.2.9 [Artificial Intelligence] Robotics — Manipulators; I.3.5 [Computer Graphics] Computational Geometry and Object Modelling — Curve, surface, solid, and object representation, — Geometric algorithms, languages, and systems, — Hierarchy and geometric transformations

General Terms: Algorithms, Theory

Keywords and phrases: quaternion, rotation, spherical geometry, spline, quadrangle spline, animation, interpolation, approximation, in-betweening, tracking

1. Introduction

The most straight-forward motion of a rigid body with one point fixed is a rotation. Possible positions of the body with respect to the point are called *orientations*. Since every orientation can be achieved by some rotation about the fixed point, and since the fixed point can be placed anywhere in space with a translation, any free position of the body in space can be described as a translation plus a rotation. (Hence it is convenient—and common—to blur the distinction between rotation and orientation, and between translation vector and point.) To animate such a body with a computer, we need numerical descriptions—i. e. coordinates—for translations and rotations. These coordinates should be chosen in such a way that simple motion is simple to describe. In particular, a steady rotation should be simple, regardless of the axis chosen, and regardless of the initial orientation. Traditional notations, such as matrices, Euler angles, or axis-angle values, fail on one or more of these criteria. In contrast, unit quaternions are consistently well-behaved.

As discussed in [16], the unit quaternions $w + \mathbf{i}x + \mathbf{j}y + \mathbf{k}z$, with $w^2 + x^2 + y^2 + z^2 = 1$, are arguably the most natural coordinates for rotations. They have the geometric structure of a sphere in four dimensions; and any steady rotation, regardless of axis and initial orientation, is obtained by steady travel along a great arc of that sphere. Simultaneously, they form an algebraic system in which the product of two quaternions gives the combination of their respective rotations. (In this they are like complex numbers of unit modulus, which can be plotted as points on a circle, multiplied with each other, and regarded as rotations in a plane.) Though sometimes peculiar, calculations with quaternions are fast and simple. Stuelpnagel[17] concludes that no other parameterization of rotations works as well. That conclusion is generally accepted in the aerospace industry[9].

Having chosen our coordinate space, we must now construct curves in it. In [16], I showed how unit quaternions can be interpolated to animate rotation, in the context of key frame 3-D animation. Duff[3] merged those ideas with curved surface rendering techniques using B-splines, to create a fast approximating curve with second-order continuity. Second-order continuity is indeed desirable; unfortunately one sacrifices the easy direct control of interpolation to get it. Either that, or the curve no longer has local control, which is unacceptable. If speed of computation were not an issue, we could use another approach: Gabriel and Kajiya[4] present an erudite—and expensive—formulation of cubic splines on a hypersphere. Their high-order, non-linear, multi-dimensional differential equation offers all the virtues of familiar splines, if one can afford to solve the necessary boundary value problem. In practice, one can't.

This paper goes deep into the mathematics of quaternions, and shows how to improve the efficiency of quaternion calculations in general, and interpolation in particular. Section 2 reviews properties of orientations, leading to the choice of quaternions as coordinates. Section 3 discusses quaternion multiplication, the foundation for everything else. Fast, unexpected implementations emerge. Section 4 explores how multiplication carries over into a representation of rotations, the use of most interest. Section 5 looks at "linear interpolation" on the unit hypersphere, and ways to make it faster. Section 6 deals with a new spherical interpolation technique that delivers tangent continuity with only three arc divisions per point, the best possible. In Section 7, quaternion differentiation reveals the required end conditions. Section 8 discusses alternative ways to control orientation, and Section 9 draws conclusions. Finally, the Appendix gives essential C routines. The emphasis throughout is on elegant mathematics efficiently applied.

2. Orientation

Good notation is a mighty lever. If we compare decimals to Roman numerals, we find decimals a great deal easier for calculation. Likewise, observe how hard it is to describe a general straight line in polar coordinates; the linear Cartesian coordinates x, y, z are so effective we forget we have a choice. Many computer animators have used Euler angle coordinates for orientation, not realizing there is a better choice. The hidden assumption in using Euler angles is that rotations act just like translations; but they don't!

Suppose you hold your arm extended and point at different objects around you. Your arm rotates around the fixed point of your shoulder, and your fingertip moves on the surface of an imaginary sphere in three-dimensional space, with your arm as radius. So directions in space are organized like a sphere, not a plane; and there is no good way to describe a sphere using Cartesian coordinates. Maps of the Earth demonstrate this, unavoidably tearing and stretching the truth.

Orientations (as opposed to directions) are actually more like a hypersphere, a sphere in four-dimensional space, because while pointing in some direction you can still rotate your wrist. Here's an interesting way to explore rotation space: Start with your finger pointing straight forward, palm down. Without rotating your wrist, swing your arm to point up, then to point to the side, then swing forward again. You'll find your palm is now facing sideways, as though you'd just rotated your wrist without moving your arm at all. This twist is a property of the way orientations connect (their topology), not a trick of anatomy.

The precise topology of orientations is that of real three-dimensional projective space, P^3, which is just slightly different from the hypersphere in four-dimensions, S^3, but very different from three-dimensional Euclidean space[17]. The hypersphere covers P^3 doubly, with each two opposite (i.e. antipodal) points on S^3 covering a single point of the projective space[12, p. 352]. Furthermore, S^3 covers the group and metric structures of orientations (actually, rotations), not just the topological structure[1, p. 20].

The practical import of the preceding abstractness is that orientations should be described using *homogeneous* coordinates, which were invented especially for projective space. These should have the form $[x, y, z, w]$, with $w^2 + x^2 + y^2 + z^2 = 1$. Also, there should be some way to multiply and divide these quadruples which will confine products to the hypersphere.

Quaternions $(w + \mathbf{i}x + \mathbf{j}y + \mathbf{k}z)$ may be considered homogeneous coordinates for orientations, which can be multiplied as required above.

3. Multiplication

Quaternions differ from reals in having three imaginary units, **i**, **j**, and **k**, with $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1$. Imaginary products do not commute: $\mathbf{ij} = \mathbf{k} = -\mathbf{ji}$, $\mathbf{jk} = \mathbf{i} = -\mathbf{kj}$, $\mathbf{ki} = \mathbf{j} = -\mathbf{ik}$. Reals are the center of the multiplication group; they commute with everything. These specific cases combine in the general rule:

$$\begin{aligned} qq' = \; & (w + \mathbf{i}x + \mathbf{j}y + \mathbf{k}z)(w' + \mathbf{i}x' + \mathbf{j}y' + \mathbf{k}z') \\ = \; & (ww' - xx' - yy' - zz') \\ & + \mathbf{i}(xw' + wx' - zy' + yz') \\ & + \mathbf{j}(yw' + zx' + wy' - xz') \\ & + \mathbf{k}(zw' - yx' + xy' + wz') \end{aligned}$$

We can now make an important observation: The product is linear in both factors. That is, $p(\alpha q + \beta q') = \alpha pq + \beta pq'$ and $(\alpha q + \beta q')p = \alpha qp + \beta q'p$, where p is a quaternion and α and β are real. One practical impact is that we can use a 4×4 matrix routine (or hardware) to multiply quaternions. Use either

$$L_q\, q' = \begin{pmatrix} w & -x & -y & -z \\ x & w & -z & y \\ y & z & w & -x \\ z & -y & x & w \end{pmatrix} \begin{pmatrix} w' \\ x' \\ y' \\ z' \end{pmatrix}$$

or

$$R_{q'}\, q = \begin{pmatrix} w' & -x' & -y' & -z' \\ x' & w' & z' & -y' \\ y' & -z' & w' & x' \\ z' & y' & -x' & w' \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix}.$$

Note that when q is of unit length L_q is an orthogonal matrix, and similarly for q' and $R_{q'}$. (An orthogonal matrix is one whose columns (equivalently, rows) are mutually perpendicular unit length vectors.) The L matrices form an algebraic system equivalent to quaternions; for example, $p(q + q')$ maps to $L_p(L_q + L_{q'})$. I mention this mainly to help understand the non-commutativity of quaternions. In practice, the matrices would be four times as expensive (e.g., $L_p\, L_q$ versus $L_p\, q$).

Associativity of quaternion multiplication makes mixing L and R matrices interesting. Because $(qp)q' = q(pq')$, we know that $R_{q'}\, L_q\, p = L_q\, R_{q'}\, p$. When p is varying but q and q' are not, either of these arrangements factors out the static part of the product for greater efficiency[10]. That is, when we have a long chain of multiplies, $q_1 q_2 \ldots q_n$, and only some inner quaternion q_i is being varied, we can collapse all left products $q_1 \ldots q_{i-1}$ into an L matrix, all right products $q_{i+1} \ldots q_n$ into an R matrix, and multiply the L and R matrices into one, leaving just a single matrix multiply for each update of q_i. Articulated objects are a likely candidate for this optimization.

The most startling discovery is that the product qq' can be computed with only 8 real multiplies[8], rather than the obvious 16. The catch is, more adds are used. Here's one way to form the product $q'' = qq'$.

$$\begin{array}{ll}
X_1 \leftarrow (z-y)\times(y'-z') & X_5 \leftarrow (z-x)\times(x'-y') \\
X_2 \leftarrow (w+x)\times(w'+x') & X_6 \leftarrow (z+x)\times(x'+y') \\
X_3 \leftarrow (w-x)\times(y'+z') & X_7 \leftarrow (w+y)\times(w'-z') \\
X_4 \leftarrow (z+y)\times(w'-x') & X_8 \leftarrow (w-y)\times(w'+z')
\end{array}$$

$$\begin{aligned}
s &\leftarrow X_6+X_7+X_8 \\
t &\leftarrow (X_5+s)/2
\end{aligned}$$

$$\begin{aligned}
w'' &\leftarrow X_1+t-X_6 \\
x'' &\leftarrow X_2+t-s \\
y'' &\leftarrow X_3+t-X_8 \\
z'' &\leftarrow X_4+t-X_7
\end{aligned}$$

4. Rotation

A quaternion can be interpreted in many ways. First, as an algebraic quantity, $w+\mathbf{i}x+\mathbf{j}y+\mathbf{k}z$. Second, as a point in 4-space, with coordinates (w,x,y,z), equivalent to homogeneous coordinates for a point in (projective) 3-space. Third, as a linear transformation of 4-space, given by left or right quaternion multiplication. Fourth, as a scalar plus a 3-vector, $w+\mathbf{v}$, where $\mathbf{v}=\mathbf{i}x+\mathbf{j}y+\mathbf{k}z$. Note that $\mathbf{v}\mathbf{v}'=-\mathbf{v}\cdot\mathbf{v}'+\mathbf{v}\times\mathbf{v}'$; dot and cross products of vectors were discovered as parts of the quaternion product.

Finally, a quaternion describes a rotation of 3-space. We can write a unit quaternion as $\cos\theta+\hat{\mathbf{v}}\sin\theta$, where $\hat{\mathbf{v}}$ is a unit vector. If we then form the product $qq'q^{-1}$, we find that the action on the vector $\mathbf{v}'$ of q', is to rotate it by 2θ around axis $\mathbf{v}$, leaving w' unchanged. (Richardson[14] omits the factor 2, hence all his conversion formulae are incorrect.) We'll also find that any scalar multiple of q acts just like q, as we would expect of homogeneous coordinates.

Proving that $q\mathbf{v}'q^{-1}$ simply rotates $\mathbf{v}'$ is instructive; we pick up both insight and notation. First, notation. The *conjugate* of $q=w+\mathbf{v}$ is $q^*=w-\mathbf{v}$. The *norm* of q is $\mathrm{N}(q)=qq^*=q^*q=w^2+x^2+y^2+z^2$, which is real. Multiplication confirms that $q^{-1}=q^*/\mathrm{N}(q)$; also that $(pq)^{-1}=q^{-1}p^{-1}$. Some computation shows the important property† $N(pq) = N(p)N(q)$. These facts imply that $(pq)^* = q^*p^*$, which can be verified for the units $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$.

The proof of rotation logically falls into two parts: First show that a vector remains a vector, then that the effect is a rotation. So, let $S(q) = (q + q^*)/2$, which extracts the scalar w. Then we can deduce $S(q\mathbf{v}'q^{-1}) = 0$, as follows. By definition, $2\,S(q\mathbf{v}'q^{-1}) = q\mathbf{v}'q^{-1} + (q\mathbf{v}'q^{-1})^*$. By the commutativity of reals, the second term is the negative of the first: $(q\mathbf{v}'q^{-1})^* = (q^{-1})^*(\mathbf{v}')^*q^* = q/N(q)(-\mathbf{v}')q^* = -q\mathbf{v}'q^{-1}$. Hence a vector remains a vector. Furthermore, $N(q\mathbf{v}'q^{-1}) = N(\mathbf{v}')$, and multiplication is bilinear. This is enough to prove we have an orthogonal transform of 3-space, i. e. a rotation and/or a reflection. We can rule out the possibility of a reflection with a simple continuity argument. The norm of q is manifestly irrelevant, so assume it is 1. Then when θ of q is 0, the transform is the identity, a trivial rotation. As θ increases, q sweeps out a great circle arc, and the transform continuously evolves through a series of rotations, until at $\theta = \pi$ we have $q = -1$, which again gives the identity transform. We have confirmed not only that $q\mathbf{v}'q^{-1}$ must effect a rotation, but also that the mapping from quaternions to rotations is two to one ($-q$ and q give the same rotation).

The product of two quaternions clearly maps to the product of their rotations, since $(pq)\mathbf{v}'(pq)^{-1} = p(q\mathbf{v}'q^{-1})p^{-1}$. And [16] proves that the vector $\mathbf{v}$ of q is the axis of the rotation q produces, while w is cosine of twice the angle of rotation. For general (non-unit) q, the corresponding 3×3 rotation matrix is

$$\frac{1}{w^2+x^2+y^2+z^2}\begin{pmatrix} w^2+x^2-y^2-z^2 & 2xy-2wz & 2xz+2wy \\ 2xy+2wz & w^2-x^2+y^2-z^2 & 2yz-2wx \\ 2xz-2wy & 2yz+2wx & w^2-x^2-y^2+z^2 \end{pmatrix}$$

This is an opportune place to comment on some implementation issues. The formula for rotation is $q\mathbf{v}'q^{-1} = L_q R_{q^{-1}} \mathbf{v}'$. When q is a unit quaternion, this suggests an easy way to convert q to a homogeneous rotation matrix: Multiply the simple matrices L_q and R_{q^*} into your current transformation. Of course, their first row

† After failing for 15 years to find a way to multiply triples so that the norm was preserved, Hamilton suddenly realized in October of 1843 that quadruples would work. He says "I then and there felt the galvanic circuit of thought *close*; and the sparks which fell from it were the *fundamental equations between i, j, k; exactly such* as I have used them ever since." [7, p. 293]

and column must be permuted to last, since computer graphics convention places the homogeneous coordinate w last. The desired 4×4 matrices are

$$\mathcal{L}_q \mathcal{R}_{q^*} = \begin{pmatrix} w & -z & y & x \\ z & w & -x & y \\ -y & x & w & z \\ -x & -y & -z & w \end{pmatrix} \begin{pmatrix} w & -z & y & -x \\ z & w & -x & -y \\ -y & x & w & -z \\ x & y & z & w \end{pmatrix}$$

Aside from numerical problems, these same matrices work even when $\mathrm{N}(q) \neq 1$. Those lacking matrix hardware may consult the appendix for an efficient conversion of a non-unit quaternion to a 3×3 matrix.

To convert from matrix to quaternion, a scheme like that of Shepperd[15] has proven numerically robust and efficient. From the diagonal entries and their sum (the *trace* of the matrix), he determines which of w^2, x^2, y^2, or z^2 is the largest, and uses that element to extract the others. Although [15] doesn't mention it, if the trace is greater than zero, using w^2 is accurate enough. This leads to the slightly faster algorithm in the appendix.

5. Slerp

Since a great arc is the spherical equivalent of a line, and since linear interpolation is commonly written *Lerp*, it seems reasonable to call interpolation on a sphere *Slerp*. There are several equivalent ways of expressing Slerp, the most symmetric of which is

$$\mathrm{Slerp}(p, q; \alpha) = \frac{p \sin(1-\alpha)\Omega + q \sin \alpha\Omega}{\sin \Omega},$$

where the 4-space dot product of p and q defines $\cos \Omega$. As α sweeps from 0 to 1, Slerp sweeps out a great arc from p to q. Slerp can be expressed more concisely in four equivalent exponential forms. In this context, q^{-1} is the same as q^*, because $\mathrm{N}(q)$ must equal 1 for Slerp to be applicable.

$$\mathrm{Slerp}(p, q; \alpha) = p(p^{-1}q)^{\alpha} = (pq^{-1})^{1-\alpha}q = (qp^{-1})^{\alpha}p = q(q^{-1}p)^{1-\alpha}$$

Since quaternion multiplication is not commutative, the equivalence of these forms is not obvious; it can be proved from the following quaternion version of Euler's identity. Noting that a unit vector $\hat{\mathbf{v}}$ squares to -1 because the cross product vanishes and the dot product is 1, we can show that Euler's identity holds for quaternions in the form $\cos\theta + \hat{\mathbf{v}}\sin\theta = \exp(\hat{\mathbf{v}}\theta) = 1 + \hat{\mathbf{v}}\theta + (\hat{\mathbf{v}}\theta)^2/2! + (\hat{\mathbf{v}}\theta)^3/3! + \cdots$. This leads to the identity $q^\alpha = \cos\alpha\theta + \hat{\mathbf{v}}\sin\alpha\theta$, from which we see that $pq^\alpha p^{-1} = \exp(p\hat{\mathbf{v}}\theta p^{-1}\alpha) = (pqp^{-1})^\alpha$. (It is important not to assume too much, however; non-commutativity implies that $(pq)^\alpha$ does *not* equal $p^\alpha q^\alpha$, nor does e^{p+q} equal $e^p e^q$, nor does $\ln(pq)$ equal $\ln(p) + \ln(q)$. These familiar identities hold only when the cross product of the vector parts of p and q vanishes.)

As a practical algorithm, Slerp looks much slower than Lerp. If the endpoints vary, about the best improvement is to use table lookup for sin and arccos. However, when α is $1/2$, we could just add p and q and normalize the result. If we have static p and q with a steadily varying α, other improvements are possible. We need only calculate Ω and $1/\sin\Omega$ once; and we can compute $\alpha\Omega$ and $(1-\alpha)\Omega$ with forward differencing. We can also pre-scale p and q by $1/\sin\Omega$. This reduces the inner loop to 8 multiplies, 6 adds, and 2 table lookups per Slerp evaluation. (By representing Ω as a fraction of 2π, these calculations can be done in fixed-point arithmetic with all values between -1 and $+1$.) All that's left is to begin cheating: When Ω is small, substitute Lerp for Slerp. In fact, this is necessary anyway, to avoid divides by nearly zero. We can just push the crossover as high as possible without getting caught. Still the moral for speed is "Minimize Slerps."

6. Spline

Key frame 3-D animation is typically based on splines made of cubic polynomial segments. This is because a cubic polynomial has four degrees of freedom: two to match first derivatives, and two to match either second derivatives or positions. As [16] makes clear, a spherical analog of cubic polynomials is needed for well-behaved curves interpolating orientations. Boehm[2], in comparing different geometric controls for cubic polynomial segments, describes an evaluation method using "quadrangle points" which requires only 3 Lerps, half the number needed for the Bézier method adapted in [16].

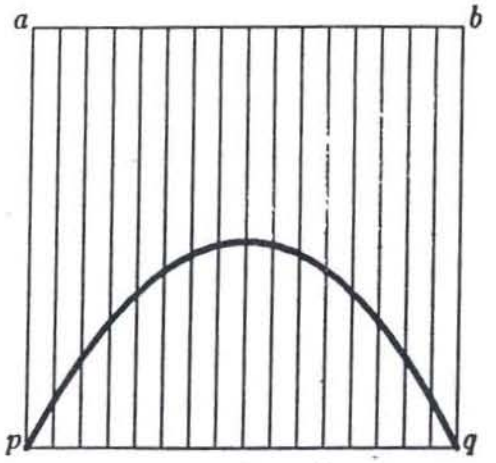


Figure 1. Square Quadrilateral

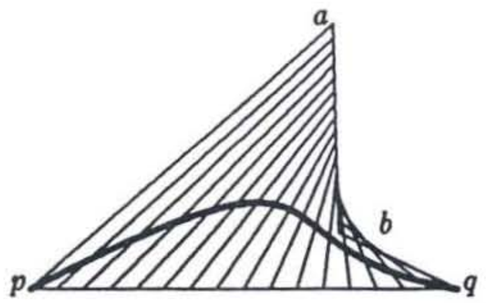


Figure 2. Warped Quadrilateral

The interpretation of this algorithm is simple: p and q form one side of a quadrilateral, a and b the opposite side; the sides may be non-parallel and non-coplanar. The two inner Lerps find points on those sides, then the outer Lerp finds a point in between. Essentially, a simple parabola drawn on a square is subjected to an arbitrary bi-linear warp, which converts it to a cubic. Transliterated into Slerps, Boehm's algorithm gives a spherical curve,

$$\mathrm{Squad}(p,a,b,q;\alpha) = \mathrm{Slerp}(\mathrm{Slerp}(p,q;\alpha), \mathrm{Slerp}(a,b;\alpha); 2(1-\alpha)\alpha).$$

As with Bézier control points, the quaternions a and b control the tangents at the end points p and q, but do so less simply. To find out exactly how to control the tangents (since we want to match them across segments), we must differentiate Squad. I'll discuss that in the next section.

Where efficiency is concerned, there are two good reasons to recommend this approach. First, only three Slerps are used. No fewer would suffice to control both end points and tangents independently. Second, two of the three Slerps have static end points, allowing the optimizations mentioned above. In contrast, the Bézier approach involves dynamic end points for half of its six Slerps. We thus expect a naive implementation of Squad to be twice as fast, and a clever implementation even faster. Still, there is no denying that forward differencing would be faster yet, if we could use it. We *can* use it if the points to be interpolated are close enough together; an ordinary matrix spline[3] won't stray too far from the unit hypersphere in that case. To generate close points, evaluate Squad for moderate steps of α. Matrix splines need only four adjacent points, which can easily be computed on demand; so this scheme is simpler than Duff's B-spline bisection[3].

7. Differentiation

Now, about those tangents. Few people today realize that Maxwell, in his classic *Treatise on Electricity and Magnetism*, used quaternion differentials[7, p. 316]; we could hardly follow in finer footsteps. Differentiating Squad is not intractable, however we ought first understand how to differentiate Slerp. This is best done in exponential form. In general, the derivative of a quaternion expression is much the same as that of a similar real expression; the main difficulty is preserving the order of multiplication. It is not too hard to show[5, p. 453] that the differential of q^{α}, where α is a real expression, is $q^{\alpha}\ln(q)d\alpha + \alpha q^{\alpha-1}dq$. (A gentler guide to quaternion calculus is [6].) Of course by Euler's identity, $\ln(q)$ is just $\hat{\mathbf{v}}\theta$ when $q = \cos\theta + \hat{\mathbf{v}}\sin\theta$. Thus we find ∂_{α}, the derivative with respect to α, of Slerp (writing Slerp as $p(p^{-1}q)^{\alpha}$) is

$$\begin{aligned}\partial_{\alpha}\ \mathrm{Slerp}(p,q;\alpha) &= p(p^{-1}q)^{\alpha}\ln(p^{-1}q)\\ &= \mathrm{Slerp}(p,q;\alpha)\ \ln(p^{-1}q)\end{aligned}.$$

In deriving the tangent of Squad at $\alpha = 0$, intermediate expressions can often be greatly simplified since we know products with α will be 0, and exponentials will be 1. (The $\alpha = 1$ case is symmetrical, and need not be worked out separately.) Also it is helpful to use coordinates where p is the identity. Thus, let $Q = p^{-1}q$, $A = p^{-1}a$, and $B = p^{-1}b$, so that

$$\mathrm{Squad}(p,a,b,q;\alpha) = p\,Q^{\alpha}(Q^{-\alpha}A(A^{-1}B)^{\alpha})^{2(1-\alpha)\alpha}.$$

Finally, let $C = A^{-1}B$, and proceed, evaluating at $\alpha = 0$.

$$
\begin{aligned}
\partial_\alpha \operatorname{Squad}(p, a, b, q; \alpha) =& p\, \partial_\alpha Q^\alpha (Q^{-\alpha} A C^\alpha)^{2(1-\alpha)\alpha} \\
=& p\, (\partial_\alpha Q^\alpha)(Q^{-\alpha} A C^\alpha)^{2(1-\alpha)\alpha} \\
& + p\, Q^\alpha\, \partial_\alpha (Q^{-\alpha} A C^\alpha)^{2(1-\alpha)\alpha} \\
=& p\, Q^\alpha \ln(Q) + p\, \partial_\alpha (Q^{-\alpha} A C^\alpha)^{2(1-\alpha)\alpha} \\
=& p \ln(Q) \\
& + p\, (Q^{-\alpha} A C^\alpha)^{2(1-\alpha)\alpha} \ln(Q^{-\alpha} A C^\alpha)\, \partial_\alpha\, 2(1-\alpha)\alpha \\
& + p\, 2(1-\alpha)\alpha (Q^{-\alpha} A C^\alpha)^{2(1-\alpha)\alpha - 1}\, \partial_\alpha (Q^{-\alpha} A C^\alpha) \\
=& p \ln(Q) + 2p \ln(Q^{-\alpha} A C^\alpha) \\
=& p \ln(Q) + 2p \ln(A) \\
=& p\, (\ln(p^{-1} q) + 2 \ln(p^{-1} a))
\end{aligned}
$$

This final expression has a fairly intuitive meaning using simple ideas from differential geometry[13, p. 146]. At every point p of the quaternion unit hypersphere, there is a 3-D linear space of tangent vectors. The logarithms give tangent space vectors for the arcs from p to q and from p to a; the weighted sum gives the desired tangent space vector; and the initial factor positions the space of that vector at the start of the curve, p. Look back at the derivative of Slerp to see how this interpretation works in that simple case. Here's an illustration of a tangent space on a sphere in three dimensions, where the tangent space is just a plane.

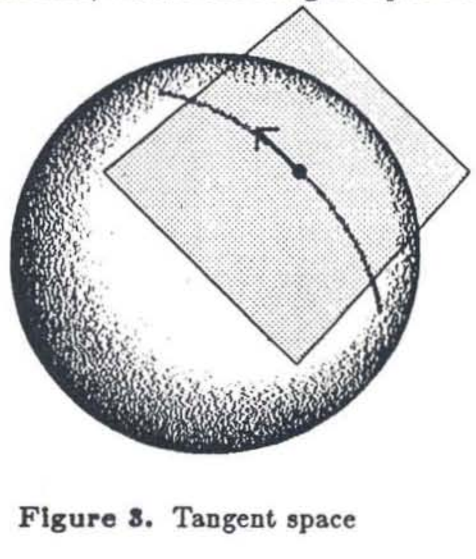


Figure 8. Tangent space

Given a series of quaternions q_n, use of Squad requires filling in values a_n and b_n on both sides of the interpolation points, so that each "cubic" segment is traced out by

Squad$(q_n, a_n, b_{n+1}, q_{n+1}; \alpha)$, as α sweeps from 0 to 1. If we supply a tangent space vector $\mathbf{t}_n$, we can solve for a_n, and by symmetry b_n, as follows:

$$a_n = q_n \exp\left(\frac{\mathbf{t}_n - \ln(q_n^{-1} q_{n+1})}{2}\right)$$
$$b_n = q_n \exp\left(\frac{-\mathbf{t}_n - \ln(q_n^{-1} q_{n-1})}{2}\right).$$

A good value for $\mathbf{t}_n$ averages the tangents of arcs to adjacent points:

$$\mathbf{t}_n = \frac{\ln(q_n^{-1} q_{n+1}) - \ln(q_n^{-1} q_{n-1})}{2}$$

so that the values for a_n and b_n are given by

$$a_n = b_n = q_n \exp\left(-\frac{\ln(q_n^{-1} q_{n+1}) + \ln(q_n^{-1} q_{n-1})}{4}\right).$$

8. Alternatives

For controlling a freely moving object or camera, Squad does very well. However it cannot be expected to simulate physical equations of motion nor to cause a camera to track an interesting object. As it happens, given an angular momentum vector $\omega(t)$, the quaternion differential equation of motion is linear, and easy to integrate[17]. This is one of the reasons quaternions have been so popular for control of spacecraft.

Tracking is another matter altogether. A vector from the eyepoint to the object tracked is typically used to specify a direction of gaze. However, as noted earlier, orientations are more than directions. In order to completely specify an orientation, the "roll" degree of freedom must be pinned down. Usually this is done by specifying a vector, some non-zero "up" component of which is perpendicular to the direction of gaze. There is, however, a fundamental difficulty with this procedure.

A theorem of differential topology says that it is impossible to construct a smooth tangent field on the sphere S^2 that does not vanish at some point[11, p. 30][12, p.

367]. Because the "up" vectors must be perpendicular to the gaze, they are tangent to the sphere of directions. If "up" for a particular direction does not depend on the path of the gaze over time, then the "up" vectors form a non-zero tangent field, which cannot be smooth everywhere.

Ordinarily the discontinuities are placed at the north and south poles, and experienced camera animators try to avoid looking nearly straight up or down. However it sometimes proves awkward to automatically avoid the neighborhoods of static discontinuities. In this case abrupt rolls must be filtered out. The result will be a more gradual roll, which does not preserve "up." For example, when tracking an object flying directly overhead, the orientation snaps abruptly through 180 degrees at the pole. Smoothing the roll spreads this change over time.

9. Conclusions

The spherical version of Boehm's quadrangle spline allows interpolation of rotations with only three Slerps per point, the minimum possible. I have shown how this adaptation, Squad, can be accomplished. I also provided a variety of means by which the cost of Slerps can be lowered, as can the cost of converting quaternions to matrices, and the cost of multiplying quaternions. Duff[3] has suggested subdividing a spherical uniform B-spline until reasonably flat, then using non-Slerp algorithms. A similar approach will further speed up Squad. Note that both schemes place keys at uniformly spaced times, depending on a monotonic time map for more flexible control. When the extra smoothness of second order continuity is less important than direct control, use Squad. For spherical interpolation with first order continuity, Squad is an efficient algorithm, and about the best that can be hoped for.

10. Acknowledgements

Warren Robinett criticized early drafts in detail, for which I am most grateful. Jules Bloomenthal contributed a fresh perspective when my vision had become stale.

11. References

[1] J. G. Belifante, B. Kolman, and H. A. Smith, "An Introduction to Lie Groups and Lie Algebras, with Applications," *SIAM Review,* **8**(1),11–46, January 1966.

[2] W. Boehm, "On Cubics: A Survey," *Comp. Graph. and Image Proc.* , **19**,201–226, 1982.

[3] T. Duff, "Splines in Animation and Modelling," in Course Notes for "State of the Art in Image Synthesis," SIGGRAPH Conference, August 1986.

[4] S. A. Gabriel and J. T. Kajiya, "Spline Interpolation in Curved Space," in Course Notes for "State of the Art in Image Synthesis," SIGGRAPH Conference, July 1985.

[5] Sir W. R. Hamilton, *Elements of Quaternions, Volume I, Third Edition,* Chelsea Publishing Co., New York, 1969.

[6] Sir W. R. Hamilton, *Lectures on Quaternions,* Hodges and Smith, Dublin, 1853.

[7] T. L. Hankins, *Sir William Rowan Hamilton,* The Johns Hopkins University Press, 1980.

[8] T. D. Howell and Jean-Claude Lafon, "The Complexity of the Quaternion Product," Cornell Comp. Sci. Dept. Tech. Rep. TR–75–245, June 1975.

[9] P. C. Hughes, *Spacecraft Attitude Dynamics,* John Wiley and Sons, Inc., New York, 1986.

[10] B. P. Ickes, "A New Method for Performing Digital Control System Attitude Computations Using Quaternions," *AIAA Journal,* January 1970.

[11] J. W. Milnor, *Topology from the Differentiable Viewpoint,* The University Press of Virginia, 1969

[12] J. R. Munkres, *Topology: A First Course,* Prentice-Hall Inc., Englewood Cliffs, NJ, 1975

[13] B. O'Neill, *Elementary Differential Geometry,* Academic Press, New York, 1966

[14] D. S. Richardson, "Quaternion Algebra for Three-Dimensional Computer Graphics and Modelling," *Australian Comp. Sci. Communications,* **5**(1),109–120, February 1983.

[15] S. W. Shepperd, "Quaternion from Rotation Matrix," *J. Guid. Control,* **1**(3),223–224, 1978.

[16] K. Shoemake, "Animating Rotation with Quaternion Curves," *Computer Graphics,* **19**(3),245–254, SIGGRAPH Conference Proceedings, July 1985.

[17] J. Stuelpnagel, "On the Parameterization of the Three-Dimensional Rotation Group," *SIAM Review,* **6**(4),422–430, October 1964.

12. Appendix

The following C routines implement essential operations described in the text. I have chosen to represent quaternions as arrays of four **double**'s, but a **struct** might be used instead. My rationale is that no questionable casts are necessary when performing vector operations (for example, dot products), and translation to other languages (like FORTRAN) is easy. On the other hand, it would be convenient to have functions return a quaternion value, which is only possible with the **struct** representation. So it goes.

```
#define X  0
#define Y  1
#define Z  2
#define W  3

#define EPSILON  0.00001
#define HALFPI  1.570796326794895
```

```
/*
 * qmul: Compute quaternion product qq = qL * qR.
 * Requires qL and qR to be distinct storage from qq.
 */
void qmul(qL,qR,qq)
double qL[4],qR[4],qq[4];
{
    qq[W] = qL[W]*qR[W] - qL[X]*qR[X] - qL[Y]*qR[Y] - qL[Z]*qR[Z];
    qq[X] = qL[W]*qR[X] + qL[X]*qR[W] + qL[Y]*qR[Z] - qL[Z]*qR[Y];
    qq[Y] = qL[W]*qR[Y] + qL[Y]*qR[W] + qL[Z]*qR[X] - qL[X]*qR[Z];
    qq[Z] = qL[W]*qR[Z] + qL[Z]*qR[W] + qL[X]*qR[Y] - qL[Y]*qR[X];
}

/*
 * qcnj: Conjugate quaternion.
 */
void qcnj(q,qq)
double q[4],qq[4];
{
    qq[X] = -q[X];
    qq[Y] = -q[Y];
    qq[Z] = -q[Z];
    qq[W] = q[W];
}

/*
 * qinv: Invert quaternion.  That is, form its multiplicative inverse.
 */
void qinv(q,qq)
double q[4],qq[4];
{
    double norminv;

    norminv = 1.0 / (q[X]*q[X] + q[Y]*q[Y] + q[Z]*q[Z] + q[W]*q[W]);
    qq[X] = -q[X] * norminv;
    qq[Y] = -q[Y] * norminv;
    qq[Z] = -q[Z] * norminv;
    qq[W] = q[W] * norminv;
}
```

```
/*
 * qexp: Exponentiate quaternion, assuming scalar part 0.
 */
void qexp(q,qq)
double q[4],qq[4];
{
    double theta,scale;

    theta = sqrt(q[X]*q[X] + q[Y]*q[Y] + q[Z]*q[Z]);
    scale = 1.0;
    if (theta > EPSILON)
        scale = sin(theta)/theta;
    qq[X] = scale*q[X];
    qq[Y] = scale*q[Y];
    qq[Z] = scale*q[Z];
    qq[W] = cos(theta);
}

/*
 * qlog: Take the natural logarithm of unit quaternion.
 */
void qlog(q,qq)
double q[4],qq[4];
{
    double theta,scale;

    scale = sqrt(q[X]*q[X] + q[Y]*q[Y] + q[Z]*q[Z]);
    theta = atan2(scale,q[W]);
    if (scale > 0.0)
        scale = theta/scale;
    qq[X] = scale * q[X];
    qq[Y] = scale * q[Y];
    qq[Z] = scale * q[Z];
    qq[W] = 0.0;
}
```

```
/*
 * quattomat: Convert quaternion to 3x3 rotation matrix.
 * Quaternion need not be unit magnitude.  When it always is,
 * this routine can be simplified.
 */
void quattomat(q,mat)
double q[4];
double mat[3][3];
{
    double s,xs,ys,zs,wx,wy,wz,xx,xy,xz,yy,yz,zz;

    /* For unit q, just set s = 2.0; or set xs = q[X] + q[X], etc. */
    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[Z] * zs;
    xx = q[X] * xs;  xy = q[X] * ys;  xz = q[Z] * zs;
    yy = q[Y] * ys;  yz = q[Z] * zs;  zz = q[Z] * zs;

    mat[X][X] = 1.0 - (yy + zz);
    mat[X][Y] = xy - wz;
    mat[X][Z] = xz + wy;
    mat[Y][X] = xy + wz;
    mat[Y][Y] = 1.0 - (xx + zz);
    mat[Y][Z] = yz - wx;
    mat[Z][X] = xz - wy;
    mat[Z][Y] = yz + wx;
    mat[Z][Z] = 1.0 - (xx + yy);
}
```

```
int nxt[3] = {Y,Z,X};
/*
 * mattoquat: Convert 3x3 rotation matrix to unit quaternion.
 */
void mattoquat(mat,q)
double mat[3][3];
double q[4];
{
    double tr,s;
    int i,j,k;

    tr = mat[X][X] + mat[Y][Y]+ mat[Z][Z];
    if (tr > 0.0) {
        s = sqrt(tr + 1.0);
        q[W] = s * 0.5;
        s = 0.5 / s;
        q[X] = (mat[Z][Y] - mat[Y][Z]) * s;
        q[Y] = (mat[X][Z] - mat[Z][X]) * s;
        q[Z] = (mat[Y][X] - mat[X][Y]) * 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 = sqrt( (mat[i][i] - (mat[j][j]+mat[k][k])) + 1.0 );
        q[i] = s * 0.5;
        s = 0.5 / s;
        q[W] = (mat[k][j] - mat[j][k]) * s;
        q[j] = (mat[j][i] + mat[i][j]) * s;
        q[k] = (mat[k][i] + mat[i][k]) * s;
    }
}
```

```
/*
 * slerp: Spherical linear interpolation of unit quaternions.
 * As t goes from 0 to 1, qt goes from p to q.
 */
void slerp(p,q,t,qt)
double p[4],q[4];
double t;
double qt[4];
{
    double omega,cosom,sinom,sclp,sclq;
    int i;

    cosom = p[X]*q[X] + p[Y]*q[Y] + p[Z]*q[Z] + p[W]*q[W];
    if ( (1.0 + cosom) > EPSILON ) {
        /* usual case */
        if ( (1.0 - cosom) > EPSILON ) {
            /* usual case */
            omega = acos(cosom);
            sinom = sin(omega);
            sclp = sin((1.0 - t)*omega) / sinom;
            sclq = sin(t*omega) / sinom;
        } else {
            /* ends very close */
            sclp = 1.0 - t;
            sclq = t;
        }
        for (i = X; i <= W; i++)
            qt[i] = sclp*p[i] + sclq*q[i];
    } else {
        /* ends nearly opposite */
        qt[X] = -p[Y];  qt[Y] = p[X];
        qt[Z] = -p[W];  qt[W] = p[Z];
        sclp = sin((1.0 - t) * HALFPI);
        sclq = sin(t * HALFPI);
        for (i = X; i <= Z; i++)
            qt[i] = sclp*p[i] + sclq*qt[i];
    }
}
```

Dynamics for Everyone

Jane Wilhelms
Computer and Information Sciences
University of California, Santa Cruz 95064
415-429-2440

ABSTRACT

There is a move in computer graphics toward more correctly simulating the world being modeled in hopes of achieving more realistic and interesting still images and animation. An important component of this move is use of dynamics, i.e. considering the world as masses acting under the influence of forces and torques. Dynamics can be useful in providing inverse kinematics, constraints, collisions, and, in general, help produce realistic positions and rates of motion. However, it is computationally expensive, involved to program, and complex to control.

1. What is Dynamics and What can it Buy Us?

Dynamics refers to the description of motion as the relationship between forces and torques acting on masses. If we treat the objects modeled in computer graphics as masses and apply forces and torques to them, we can use physics to find out the motion these masses should undergo. This motion should mimic the motion that would actually occur to such masses in the real world, hence dynamics *simulates* the motion, rather than just *animating* it.

Dynamics is useful for a number of reasons: it can help restrict motion to that which is realistic in the world modeled; it can automatically find many kinds of complex motion with minimal user input (e.g., motion due to gravity); it can automatically impose many kinds of constraints (e.g., preventing intersection of colliding bodies); it can be used to move complex bodies in natural way; etc.

Dynamics is problematic as a technique for motion control in computer animation because it is (often) computationally expensive, and because controlling the motion is (often) difficult. However, it shows considerable potential for manipulating and animating bodies, and merits further investigation.

This paper attempts to provide enough basic information to let anyone simulate simple objects using dynamics. A caveat: I'm not a physicist and I haven't had everything here carefully checked by one. It is a culling of relevant information from lots of different sources, which are listed in the references at the back. I would be glad to hear about errors and suggested improvements.

2. How To Do It?

To use dynamics to find the motion of objects, first we must set up the *dynamics equations of motion* which describe how *masses* will move under the influence of *forces* and *torques*. Though there are a number of ways to formulate the equations, they all should give the same solution (they refer to the same world). Second, we must *solve* the equations for *acceleration*. Third, we must *integrate* to find the new *velocity* and *position*, given that acceleration. Once we have the new position, we can animate the object.

There are many books discussing dynamics; unless some specific reference needs to be made, most of the physics in this paper relies upon these references.[7, 10, 17, 19, 21] Robotics books are often useful.[14, 18] The following references pertaining to use of dynamics for computer animation may also be useful.[1, 2, 3, 22, 23, 24, 25, 26]

We will be assuming a right-handed coordinate system with a right-hand screw rule for rotations, and I am assuming that vectors are premultiplied by matrices to change coordinate frames. (This is more in keeping with robotics and physics usage than computer graphics.) Note that considerable variation in conventions are found in the literature; keep in mind which frame and which screw rule you are using. [14, 18]

Matrices will be in uppercase boldface type (**J**), vectors in lowercase boldface (**f**), and scalars in italic type (m). Subscripts will be used to describe the axis for vectors ($\mathbf{c}_x$

This work was supported by National Science Foundation grant number CCR-8606519 and UCSC fellowship 660177-19900.

is the position of the center of mass along the x-axis), and to further describe the value when necessary ($\mathbf{f}_{grv,i,x}$ is the force of gravity acting on the i-th segment along the x-axis). Superscripts will be used to indicate the frame of reference being used, when necessary ($\mathbf{c}^{j}_{i,x}$ is the above seen in terms of the instantaneous position of the j-th coordinate frame).

Table 1. is a handy reference for the meaning of terms.

2.1. Particles: Point Masses

To illustrate the method on a very simple object, consider the motion of a point mass (a *particle*) in three-dimensions. Dynamics can be done in two dimensions and it's much easier, but also much less interesting.

2.1.1. Information Needed

2.1.1.1. Invariant Information

The only extra piece of constant information we need to dynamically animate particles is the *mass* of the particle. (We could also do dynamics on a particle of changing mass but it's probable that, for computer graphical purposes, constant mass is a reasonable assumption.)

2.1.1.2. Variable Information

Variable data we need for dynamically animating particles includes its present position **p** (a 3d vector representing x,y, and z-coordinates) and its present velocity **v** (also a 3d vector representing the present motion of the particle). (Again, other coordinate systems could be used, but the cartesian x,y,z system seems reasonable.) The fact that we need 3 numbers to specify the position implies that the particle has three degrees of freedom of motion.

We also need to know the force **f** (a 3d vector with components pulling along the x,y, and z-axis) being applied. If a number of forces are pulling at once, we need only add the vectors representing the individual forces to get a net force.

2.1.2. Equations

According to Newton's Second Law, the dynamics of a particle can be stated as

$$\mathbf{f} = m\,\mathbf{a} \qquad 1.$$

where **f** is the force (a 3d vector representing the components of the force along each cartesian axis) acting on the particle, m is the mass of the particle, and **a** is the acceleration that the particle will undergo. Typically, force is in *Newtons* ($kilograms-meters/second^2$), mass is in *kilograms*, and acceleration is in $meters/second^2$.

This vector equation really represents three scalar equations, one for each cartesian axis. These three equations are

$$f_x = m\, a_x \tag{1.a}$$

$$f_y = m\, a_y \tag{1.b}$$

$$f_z = m\, a_z \tag{1.c}$$

The Second Law Equation is a differential equation, because the acceleration is a function of time. The equation can be also stated

$$\mathbf{f} = \mathrm{m}\frac{\mathrm{d}\,\mathbf{v}}{\mathrm{dt}} \tag{2.}$$

because the acceleration is really the derivative (rate of change) of the velocity over time. (The force may also vary with time.) Similarly, it could be stated

$$\mathbf{f} = \mathrm{m}\frac{\mathrm{d}^2\mathbf{p}}{\mathrm{dt}^2} \tag{3.}$$

because the velocity is the derivative of the position over time, and, thus, acceleration is the second derivative of the position.

2.1.3. Solving the Equations of Motion

If the user provides the particle mass and the applied force, it is easy to see that solving these three independent equations will give the acceleration that the particle will undergo along each cartesian axis, by dividing by the mass. For example, for x

$$a_x = \frac{f_x}{m} \tag{4.}$$

2.1.4. Integrating to Find the New Velocity and Position

The above equations will give us the acceleration, but not the position. A simple method of integrating this equation is referred to as the *Euler method.* It is a numerical (= approximate) solution whose inaccuracy increases as does the acceleration or the time steps used. The Euler method assumes we know the present velocity (e.g. at time i) and want to find the velocity a bit (δt) further on in time; the new velocity will be

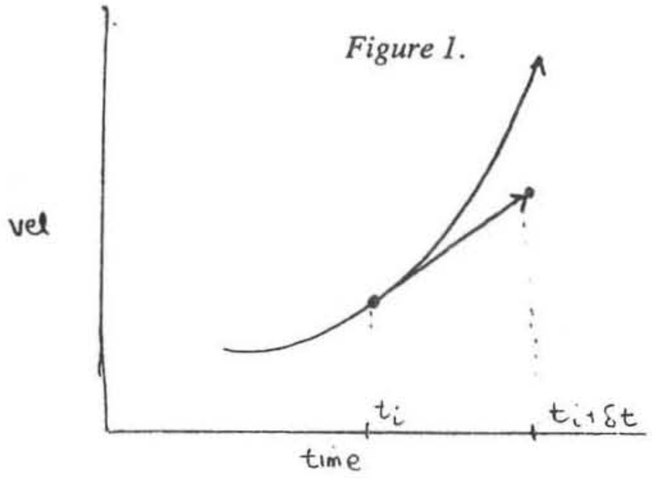
$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \delta t \tag{5.}$$

Again, this is really three separate equations. For example, for x

$$v_{i+1,x} = v_{i,x} + a_{i,x}\delta t \tag{5.a}$$

This gives us an approximation of the new velocity, but only an approximation. See Figure 1., which represents how the velocity is really changing over time. A point on the curve at time t_i represent the velocity at a particular time t_i. The arrow leaving the curve at a tangent represents the instantaneous acceleration at that time, found from Equation 4. in the previous section. The Euler approximation amounts to moving δt units along the time axis and assumes the new velocity is where the arrow is at time $t_i + \delta t$. Note this is not on the curve. How far off the curve it is depends on how much the curve is bending away from the arrow and how large δt is. With reasonably small time steps we can use this method without too much trouble arising.

Figure 1.



Given the new velocity, we can now find the new position by the same method

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \mathbf{v}_i \delta t + \frac{1}{2}\mathbf{a}_i \delta t^2 \qquad 6.$$

Again, this is really three separate equations. For x,

$$p_{i+1,x} = p_{i,x} + v_{i,x} \delta t + \frac{1}{2} a_{i,x} \delta t^2 \qquad 6.a$$

The same inaccuracy problem occurs when finding the new position. There are better methods of numerical integration, such as the Runge-Kutta method.[5]

2.1.5. Controlling the Motion

Controlling particles is pretty simple. The user need only supply an external force as one 3d vector, or as a normalized (length 1) 3d vector representing the direction of the force and a scalar magnitude representing the strength of the force. It might be desirable to have gravity act on the particle. The gravitational force $\mathbf{f}_{grv}$ is the product of a gravitational acceleration (about 9.81 *meters/second*^{2} on earth, acting toward the earth's center) times the particle mass.

Others forces that might be of interest involve collisions with other objects, and are discussed briefly later on.

2.2. Rigid Bodies: Extended Masses

Assuming that the objects are extended masses, not point masses, complicates things considerably. We assume for now that these extended masses are rigid, and do not change shape or mass.

2.2.1. Information Needed

2.2.1.1. Invariant Information

The constant information that we need includes the mass m of the object, the *center of mass* **c** of the object (the balance point), and a way to describe how the mass is distributed about the center of mass. The mass is simple.

The center of mass is a 3d vector describing a location in space. This could be a vector from the origin of the world (inertial) space within which all objects are placed, but then we would have to keep changing it as the object moved. It is better to assume some *local* coordinate frame fixed to the object and describe the center of mass relative to this local frame. As long as we know where the local frame is relative to the world frame, it is easy to find the world space center of mass if necessary. Typically such a local frame is already used to describe the geometry of objects for graphics. If the center of mass is not known, picking a point roughly at the center of the object generally is sufficient.

Describing the mass distribution can be more complex, particularly if the object is not symmetrical. Mass distribution for symmetrical objects requires three *moments of inertia*, one about each axis.

$$I_x = \int (y^2 + z^2) dm \tag{7.a}$$

$$I_y = \int (x^2 + z^2) dm \tag{7.b}$$

$$I_z = \int (x^2 + y^2) dm \tag{7.c}$$

i.e., the sum of the masses of each particle making up the object (dm) multiplied by the square of its perpendicular distance from the axis.

For symmetrical bodies there are simple ways of calculating these moments of inertia. For example, for a box centered at the origin with width c in x, b in y, and a in z, the moments of inertia around the origin are

$$I_x = \frac{1}{12} m (a^2 + b^2) \tag{8.a}$$

$$I_y = \frac{1}{12} m (a^2 + c^2) \tag{8.b}$$

$$I_z = \frac{1}{12} m (b^2 + c^2) \tag{8.c}$$

Often this bounding box is a close enough approximation.

If the object is not symmetrical, the three *products of inertia* must also be found. For objects symmetrically arranged around a center of mass, the products of inertia relative to the center of mass are all zero. The products of inertia are shown below. (Note that occasionally products of inertia are predefined as negative quantities, making terms involving them change sign in the dynamics equations.)[10]

$$I_{xy} = \int xy \; dm \qquad 9.a$$

$$I_{xz} = \int xz \; dm \qquad 9.b$$

$$I_{yz} = \int yz \; dm \qquad 9.c$$

The units for moments and products of inertia in the metric system are *kilogram−meters*^{2}.

Often the moments and products of inertia are arranged in a 3x3 *inertial tensor* matrix for using in the equations of motion

$$\mathbf{J}_k = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{xy} & I_y & -I_{yz} \\ -I_{xz} & -I_{yz} & I_z \end{bmatrix} \qquad 10.$$

Estimating the moments of inertia for simple symmetrical bodies is simple. It is also quite straightforward to find the moments and products of inertia about any axes or points in space given this information. For example, if you should want these values for the axes of a second coordinate system whose major axes are parallel to the local frame but displaced by $(\delta x, \delta y, \delta z)$, the new values are

$$I'_x = I_x + m(\delta y^2 + \delta z^2) \qquad 11.a$$

$$I'_y = I_y + m(\delta x^2 + \delta z^2) \qquad 11.b$$

$$I'_z = I_z + m(\delta x^2 + \delta y^2) \qquad 11.c$$

$$I'_{xy} = I_{xy} + m \; \delta x \; \delta y \qquad 12.a$$

$$I'_{xz} = I_{xz} + m \; \delta x \; \delta z \qquad 12.b$$

$$I'_{yz} = I_{yz} + m \; \delta y \; \delta z \qquad 12.c$$

Suppose that the new frame isn't parallel to the old. Note that this case may avoidable in your simulations, however, it is worth examining. Equations 11. and 12. take us to a new frame f' whose origin is the same as the desired rotated frame f''. Now we need to find the values for the rotated frame. To do this we need to find the *direction cosines* describing how the new x-axis is related to the old x-axis (a_{00}, a_{10}, a_{20}), the new y-axis to the old y-axis (a_{01}, a_{11}, a_{21}), and the new z-axis to the old z-axis (a_{02}, a_{12}, a_{22}).[21, 15]

We can think of the 3x3 rotation matrix representing the orientation of a frame as 3 direction cosine (column) vectors defining the axis of the frame. Column 0 represents the new x-axis, column 1 the new y-axis, and column 2 the new z-axis. To convince yourself of this relationship, try transforming the original axis vectors ((1,0,0),(0,1,0),(0,0,1)) by the rotation matrix.

$$\mathbf{D}_k = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \qquad 13.$$

Now, the new moments and products of inertia (Ix'', etc.) given those found above in a frame parallel to that centered on the center of gravity (Ix', etc.) are

$$I''_x = I'_x a_{00}^2 + I'_y a_{01}^2 + I'_z a_{02}^2 - 2I_{xy} a_{00} a_{01} - 2I_{xz} a_{00} a_{02} - 2I_{yz} a_{01} a_{02} \qquad 14.a$$

$$I''_y = I'_x a_{10}^2 + I'_y a_{11}^2 + I'_z a_{12}^2 - 2I_{xy} a_{10} a_{11} - 2I_{xz} a_{10} a_{12} - 2I_{yz} a_{11} a_{12} \qquad 14.b$$

$$I''_y = I'_x a_{20}^2 + I'_y a_{21}^2 + I'_z a_{22}^2 - 2I_{xy} a_{20} a_{21} - 2I_{xz} a_{20} a_{22} - 2I_{yz} a_{21} a_{22} \qquad 14.c$$

Similarly, the products of inertia are

$$\begin{aligned} I''_{xy} = {} & (a_{00}a_{11} + a_{01}a_{10})I'_{xy} + (a_{00}a_{12} + a_{02}a_{20})I'_{xz} + (a_{01}a_{12} + a_{02}a_{11})I'_{yz} \\ & - (a_{00}a_{10}I'_x + a_{01}a_{11}I'_y + a_{02}a_{12}I'_z) \end{aligned} \qquad 15.a$$

$$\begin{aligned} I''_{xz} = {} & (a_{00}a_{21} + a_{01}a_{20})I'_{xy} + (a_{00}a_{22} + a_{02}a_{10})I'_{xz} + (a_{01}a_{22} + a_{21}a_{02})I'_{yz} \\ & - (a_{00}a_{20}I'_x + a_{01}a_{21}I'_y + a_{02}a_{22}I'_z) \end{aligned} \qquad 15.b$$

$$\begin{aligned} I''_{yz} = {} & (a_{10}a_{21} + a_{11}a_{20})I'_{xy} + (a_{10}a_{22} + a_{20}a_{12})I'_{xz} + (a_{11}a_{22} + a_{12}a_{21})I'_{yz} \\ & - (a_{10}a_{20}I'_x + a_{11}a_{21}I'_y + a_{12}a_{22}I'_z) \end{aligned} \qquad 15.c$$

This may seem like a drastic amount of trouble, but actually it can be programmed as subroutines and made invisible to the user. In fact, approximate quantities can be found by merely providing a boundary box around the center of mass and assuming some default density to the material (e.g. 1 $kilogram / meter^3$). The dimensions of the boundary box (a, b, c) can be used to find the volume ($a \times b \times c$ $meters^3$). Multiplying the density by the volume gives the mass. The center of mass can be assumed to be the center of the bounding box. The moments of inertia around the center of mass can be found from Equation 8. above; the products of inertia will be zero. If the frame not at the center of mass but translated away from it, Equations 11. and 12. can be used to find the moments and products of inertia relative to this new frame. If the frame is rotated, Equations 13. and 14. can be used to find the new moments and products of inertia.

2.2.1.2. Variable Information

Rigid bodies have six degrees of freedom. Three are the translational degrees of freedom as with point masses. Three are rotational degrees of freedom describing how the body is oriented toward some frame of reference. Assuming a local coordinate frame fixed to the object, the translational degrees of freedom may represent displacement relative to a fixed inertial world frame axes, or along the present local frame axes (or any other axes). Similarly, the orientation degrees of freedom may refer to rotation about the world space axes, or about the present local frame axes.

We assume the order of rotations will be fixed as x-rotation, then y-rotation, then z-rotation. This means rotations are *Euler*. Euler rotations can come in various orders, here we follow the order x, then y, then z, so that the x-rotation is relative to the original x-axis, the y-rotation is about the y-axis created by the x-rotation, and the z-rotation is about the z-axis created by the former two rotations. Amazingly enough, this can also be thought of as a z-rotation, then a y-rotation, then an x-rotation around the original frame. It is often sensible to assume the local z-axis represents the longitudinal axis of the body, when there is an obvious longitudinal axis.

The other variant information involves the forces **f** and torques τ that cause motion to occur. If a number of forces are acting on the body, their total translational effect can be found by merely summing them. The center of mass of the body will move translationally as if it were a particle mass influenced by one net force.

A torque is similar to a force, except that it causes a rotational motion about a particular axis. Torques can be represented as 3d vectors describing their components about an x, y, and z-axis. Torque vectors' net action can be found by summing them.

If all forces are applied at the center of mass, they produce no torque; however, a force acting at a point on the body other than the center of mass will also cause a torque. To find a torque about a coordinate frame's axes due to a force **f** (f_x, f_y, f_z) applied at point **p** (x, y, z) (both defined relative to this frame), use this equation.

$$\tau = \mathbf{p} \times \mathbf{f} \tag{16.}$$

or, using components

$$\tau_x = f_z y - f_y z \tag{16.a}$$

$$\tau_y = f_x z - f_z x \tag{16.b}$$

$$\tau_z = f_y x - f_x y \tag{16.c}$$

Often we want motion of the rigid body in terms of its body-fixed frame, and the point of application of the force is in terms of this frame, but the external force is more naturally given in terms of the world inertial frame. An external force (or any other quantity) defined in the inertial frame can be converted into the local frame by multiplying it by the matrix defining how the world frame is oriented as seen from the local frame. This matrix is the inverse (= transpose) of the matrix defining how the local frame is defined relative to the world frame.

If multiple forces and torques are acting upon a body, these six important net values (3 force, 3 torque) can be easily found (for motion relative to the local frame) by summing the forces (in local terms) to find the net **f**, finding the torques caused by these forces using Equation 15., and summing these torques with any active pure torques to find the net torque (τ). This effectively removes the torque component from the forces. After this is done, the net force effectively is applied to the origin of the local frame. The local frame need not be at the center of mass for this to be true.

2.2.2. Equations

With rigid bodies, dynamics becomes somewhat less trivial. There are a number of formulations, and here a brief description of the Euler method is presented. The Euler method is, perhaps, one of the more intuitive formulations. The Armstrong method for articulated body dynamics presented in the next section can, of course, also be used for a single non-articulated body.

The Euler method creates six equations: three are the translational equations of motion relating the linear acceleration and mass to the force, and three are the rotational equations of motion relating the angular acceleration and mass distribution to the torque. Altogether, they specify the behavior of the six degrees of freedom of a free rigid body. Much of this discussion comes from Wells.[21]

The 3d vector version of the translational equations describing the motion of the center of mass is familiar, e.g.

$$\mathbf{f} = m\,\mathbf{a} \qquad 17.$$

or, as 3 scalar equations,

$$f_x = ma_x \;\; ; \;\; f_y = ma_y \;\; ; \;\; f_z = ma_z \qquad 17.a,b,c$$

where **f** is the net force and **a** is the linear acceleration of the center of mass relative to inertial space. This is because the center of mass acts as if the whole body mass were located there and all forces are acting at that point. The effect of these forces on rotation comes out in the rotational equations.

The force and linear acceleration could be expressed relative to any axes, e.g. the instantaneous local axis fixed to the body, by taking the proper components. However, they must both be expressed relative to the same frame. This is an important point, if the user inputs the force $\mathbf{f}^w$ relative to the inertial world coordinate frame and wants the linear acceleration $\mathbf{a}^l$ in terms of the local frame, direction cosines (= rotation matrices) can be used to find the components of the worldspace force relative to the local frame. Another way of looking at this is to take the dot product of the force vector (f_x, f_y, f_z) with each axis vector (e.g., for the x-axis, (a_{00}, a_{10}, a_{20})). The force component along the local x-axis would be

$$f_x^l = f_x^w a_{00} + f_y^w a_{10} + f_z^w a_{20} \qquad 18.$$

The rotational equations for motion about the center of mass are also quite simple, assuming the products of inertia are zero and that *either* the local frame is at the center of mass or the origin of the local frame is fixed in world space . In this case,

$$\tau_x = I_x \dot{\omega}_x + (I_z - I_y)\omega_y \omega_z \qquad 19.a$$

$$\tau_y = I_y \dot{\omega}_y + (I_x - I_z)\omega_x \omega_z \qquad 19.b$$

$$\tau_z = I_z \dot{\omega}_z + (I_y - I_x)\omega_x \omega_y \qquad 19.c$$

where all values are assumed relative to the local body-fixed frame. ω is the angular velocity of the local frame relative to the inertial frame but expressed in terms of local frame axes. ω is the angular acceleration. ω is typically in *radians/second* and ω in *radians/second*^{2}. τ is the torque acting on the body.

Should you not be so lucky, the more general form of the equations is below. All values are relative to a single coordinate frame, which may be an inertial frame, but is (for our case) probably the instantaneous position and orientation of a body-fixed local coordinate frame. **c** refers to the location of the center of mass relative to this frame. **a** refers to linear acceleration of the origin of this frame. All other values are in terms of this frame as well.

$$f_x = m(a_x - c_x(\omega_y^2 + \omega_z^2) + c_y(\omega_y \omega_x - \dot{\omega}_z) + c_z(\omega_x \omega_z + \dot{\omega}_y)) \qquad 20.a$$

$$f_y = m(a_y + c_x(\omega_x \omega_y + \dot{\omega}_z) - c_y(\omega_x^2 + \omega_z^2) + c_z(\omega_y \omega_z - \dot{\omega}_x)) \qquad 20.b$$

$$f_z = m(a_z + c_{a_x}\omega_z - \dot{\omega}_y) + c_y(\omega_y \omega_z + \dot{\omega}_x) - c_z(\omega_x^2 + \omega_y^2)) \qquad 20.c$$

$$\tau_x = m(a_z c_y - a_y c_z) + I_x \dot{\omega}_x + (I_z - I_y)\omega_y \omega_z + I_{xy}(\omega_x \omega_z - \dot{\omega}_y) - I_{xz}(\omega_x \omega_y + \dot{\omega}_z) + I_{yz}(\omega_z^2 - \omega_y^2) \qquad 21.a$$

$$\tau_y = m(a_x c_z - a_z c_x) + I_y \dot{\omega}_y + (I_x - I_z)\omega_x \omega_z + I_{yz}(\omega_y \omega_x - \dot{\omega}_z) - I_{xy}(\omega_y \omega_z + \dot{\omega}_x) + I_{xz}(\omega_x^2 - \omega_z^2) \qquad 21.b$$

$$\tau_z = m(a_y c_x - a_x c_y) + I_z \dot{\omega}_z + (I_y - I_x)\omega_x \omega_y + I_{xz}(\omega_y \omega_z - \dot{\omega}_x) - I_{yz}(\omega_x \omega_z + \dot{\omega}_y) + I_{xy}(\omega_y^2 - \omega_x^2) \qquad 21.c$$

2.2.3. Solving the Equations

Again, the Euler method of numerical integration is often adequate to solve the equations. Note the equations are simple to solve in the direct direction, given accelerations find the forces and torques; however, we want to find linear and angular accelerations given forces and torques. We assume we know the present position and velocity values. Thus we have (at worst) six equations in six unknowns ($a_x, a_y, a_z, \omega_x, \omega_y, \omega_z$).

2.2.4. Controlling the Motion

Rigid bodies can be controlled by a combination of applied torques and applied forces. Applied torques cause a rotational motion about the axes they refer to (e.g. the body-fixed local frame) and require a 3d vector. Applied forces involve a 3d force vector (as with point masses) and also a 3d location vector describing where the force is being applied. Typically the location vector will be specified in the local frame.

Net force is found by summing force vectors irrespective of point of application. Net torque is found by taking the torque caused by these forces (using Equation 15.) as well as any pure torques and summing these. These six values are used in the six equations of motion.

3. Articulated Bodies

Articulated bodies can be thought of as rigid segments connected together by joints capable of less than 6 degrees of freedom. There are numerous formulations of the dynamics equations for rigid bodies, but again, they all come down to the same thing. Some possible choices are the Euler equations,[21] the Gibbs-Appell formulation,[12, 17, 25] the Armstrong recursive formulation,[1, 3] and the Featherstone recursive formulation.[6] The Euler method doesn't deal terribly nicely with constraints at joints. The Gibbs-Appell equations, described in appalling detail elsewhere,[25] have been used for graphical simulation but in a non-recursive form that is $O(n^4)$ in complexity. This is computationally untenable, but if a recursive formulation could be found it still might be a reasonable method, as it allows considerable flexibility in designing joints. (You can design bodies that aren't a hierarchical tree structure alone.) The Featherstone method is recursive and linear in the number of joints, and is flexible in the types of joints, so it might be worth looking into.

The Armstrong method is recursive and linear in the number of joints and will be described in some detail here. It has the slight disadvantage that it can only accommodate bodies with freedom of movement relative to the world (6 degrees of freedom from the body tree root and the world) and three rotary degrees of freedom at each joint. Also bodies must be representable as tree structures. This is fine for most animalistic figures, and further constraints can be applied on top of the basic dynamics using external forces or other more devious methods. The Armstrong method has been used in graphics modeling and I am using it at present, using a modified version of code originally provided by Bill Armstrong and Mark Green at the University of Alberta.

The Armstrong method can be thought of as an extension of the Euler equations with multiple segments (connected rigid bodies). Again, there are at most six equations for each joint (one for each degree of freedom of motion). The real difference comes in the components of the torques and forces. We must consider not only applied forces and torques on the segment, but forces perculating down onto the segment from the children segments, and reaction forces at the joint between the segment and its parent. The following equations are described in detail in Armstrong and Greens 1985 paper.[3] They are repeated here in slightly different terms to show their equivalence to the Euler formulations above.

3.1. Information

The same information is needed for articulated bodies made of rigid segments as for non-articulated rigid bodies, plus a tree describing how the segments are connected together. Each segment can have at most one parent and zero or more children. For convenience, the local frame should originate at the proximal (nearer to the root) joint of a segment and the longitudinal axis of the segment should be the local z-axis. If this convention is followed, the third Euler rotation at a joint will always cause a longitudinal rotation.

If simulating people and other animals, biology and biomechanics books are useful sources of information on the nature of organic tissue, dimensions, etc. NASA's book on anthropometry is also a handy reference.[16]

3.2. Armstrong Equations

Again we have six equations, shown below as two vector equations identical to the Euler equations given above. Everything is expressed in terms of the instantaneous location and orientation of the frame of the i-th segment.

$$\mathbf{f}_i = m_i \mathbf{a}_i - m_i \mathbf{c}_i \times \dot{\omega}_i + m_i \omega_i \times (\omega_i \times \mathbf{c}_i) \qquad 22.$$

$$\tau_i = \mathbf{J}_i \dot{\omega}_i + m_i \mathbf{c}_i \times \mathbf{a}_i + \omega_i \times \mathbf{J}_i \omega_i \qquad 23.$$

In Equation 22., the first term on the right comes from the linear acceleration of frame i, the second from the angular acceleration of frame i, and the third term from the centrifugal force due to rotation of the frame. In Equation 23., the first term on the right is the rate of change of the angular momentum, the second is due to the acceleration of the frame. and the third is due to the rotation of the frame.

If the body is articulated, we must also consider the influence of neighboring segments; in any case we may want to consider external applied forces separate from gravity (pushes and pulls). We can break the force up further into

$$\mathbf{f}_i = m_i \mathbf{a}_{grv,i} + \mathbf{f}_{ext,i} + \sum \mathbf{f}_{son,i} - \mathbf{f}_{topar,i} \qquad 24.$$

All these are expressed in terms of the i-th local frame. $m\,\mathbf{a}_{grv,i}$ $(= \mathbf{f}_{grv,i})$ is the force due to gravity acting on the mass of segment i. $\mathbf{f}_{ext,i}$ is the net external force acting on frame i. $\mathbf{f}_{son,i}$ is the net force due to each son of segment i acting on segment i through the joint joining them. $\mathbf{f}_{topar,i}$ is the net force that segment i is applying to its parent. This force is applied by the parent back onto the son to keep the two from separating (as described in Newton's Third Law), so it is negative in this equation.

We can also break the torques acting on segment i into components

$$\tau_i = m_i \mathbf{c}_i \times \mathbf{a}_{grv,i} + \tau_{ext,i} + \sum(\tau_{son,i} + \mathbf{l}_{son} \times \mathbf{f}_{son}) - \tau_{topar,i} \qquad 25.$$

The first term on the right, $m_i \mathbf{c}_i \times \mathbf{a}_{grv,i}$ $(= \tau_{grv,i})$, describes the effect of gravity acting on the center of mass of the segment and causing a torque at the proximal joint. $\tau_{ext,i}$ is the net external torque applied to the segment i. $\tau_{son,i}$ is the torque that a son of segment i is applying to segment i at the joint between them. $\mathbf{l}_{son} \times \mathbf{f}_{son}$ is the torque due to the force a son segment is applying onto segment i. $\mathbf{l}_{son}$ is a vector from the origin of segment i to the joint between segment i and its son *son* in terms of frame i. $\tau_{topar,i}$ is the torque that segment i is applying to its parent segment. Forces acting directly on segment i are assumed to have been analyzed to find their torque component acting on segment i and this added to the applied external torques τ_i.

Finally, one more vector equation is needed that relates the acceleration of the parent and son segments. The right side describes the acceleration of the son's proximal hinge due to the the acceleration, angular acceleration, and centrifugal acceleration of the parent i. All are in terms of the axes of frame i.

$$\mathbf{a}_{son} = \mathbf{a}_i - \mathbf{l}_{son} \times \dot{\omega}_i + \omega_i \times (\omega_i \times \mathbf{l}_{son}) \qquad 26.$$

One thing to keep in mind is that though the motion is being described in terms of the axes of frame i, the motion is relative to inertial space, not the the parent. That is, we are not talking about the velocity relative to the parent, which may also be moving on its own. We are talking about an inertial motion that includes the motion of the segment about its joint to the parent plus any motion that parent may be involved in relative to the world.

3.3. Solving the Equations Recursively

Because we limit the body to a tree structure, effects of other segments on a particular segment is limited to effects of sons and parent on this segment. This makes it possible to solve the equations recursively. First we must recognize the linear relationship between angular and linear acceleration, and between linear acceleration and the reactive force on the parent. **K** and **M** are recursive coefficient matrices which relate linear acceleration to angular acceleration $(\dot{\omega})$ and to reactive force on the parent $(\mathbf{f}_{topar})$, respectively. **d** includes other constituents of the angular acceleration and $\mathbf{f}'$ includes other constituents of the force on the parent. For each segment i,

$$\dot{\omega}_i = \mathbf{K}_i \mathbf{a}_i + \mathbf{d}_i \qquad 27.$$

$$\mathbf{f}_{topar,i} = \mathbf{M}_i \mathbf{a}_i + \mathbf{f}'_i \qquad 28.$$

Note that the reactive force $\mathbf{f}_{topar,i}$ acting on the parent j of segment i is one of the $\mathbf{f}_{son,j}$ forces seen from this parent (see Equation 24.). By some deft maneuvering described in more detail in Armstrong and Green's 1985 paper, the dynamics equations can be restated using this relationship. The four recursive coefficients for each segment can be found in an inward pass from the leaves of the body tree to the root. Then this information can be used to find the accelerations of each segment from the root back to the leaves. The root segment has no parent, so it has no reactive force on a parent and Equation 28. can be solved for the root's linear acceleration. This can be used in Equation 27. to find the angular acceleration of the root. This process is repeated outward using the relationship in Equation 26. to find the linear acceleration of the son links and using this to find their angular acceleration.

The actual steps are shown below.[3] Note that $\mathbf{R}^{topar}$ signifies a 3x3 rotation matrix that takes vectors in a local frame into its parent frame, and $\mathbf{R}^{frompar}$ signifies a 3x3 rotation matrix that takes vectors in a parent frame into a son frame, and that these two are transposes of each other. $\mathbf{R}^{toworld}$ signifies a 3x3 rotation matrix that takes vectors in a local frame into the world frame, and $\mathbf{R}^{fromworld}$ signifies a 3x3 rotation matrix that takes vectors from the world frame into a local frame, and these two are also transposes of each other.

It is useful to compute the cross-product operation using a *tilde* matrix. The tilde matrix for a vector **a** is a 3x3 matrix that when premultiplied to a vector **b** gives the same result as the cross-product $\mathbf{a} \times \mathbf{b}$. It looks like this

$$\tilde{\mathbf{a}} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \qquad 29.$$

Inward Pass. The inward pass computes the 4 recursive coefficients and some other useful quantities that are used often. (I have slightly simplified this step. Readers are invited to find more quantities to efficiently precompute.) This step can be divided into two passes: one (the *slowband*) need only be done occasionally; the other (the *fastband*) needs to be done each time through the dynamics loop. Remember subscripts indicate which segment the value refers to, and superscripts indicate which frame the value is in terms of (unless it's in frame i). The equations are repeated for each segment. Summations are over all sons of segment i.

The slowband calculations for a segment i are these:

$$\mathbf{a}_{c,son} = \omega_i \times (\omega_i \times \mathbf{l}_{son}) \qquad 30.$$

$$\mathbf{Q}_{son} = \mathbf{R}^{topar}_{son} \mathbf{M}^{son}_{son} \mathbf{R}^{frompar}_{son} \qquad 31.$$

$$\mathbf{W}_{son} = \tilde{\mathbf{l}}_{son} \mathbf{Q}_{son} \qquad 32.$$

$$\mathbf{T}_i = (\mathbf{J}_i + \sum (\mathbf{W}_{son} \tilde{\mathbf{l}}_{son}))^{-1} \qquad 33.$$

$$\mathbf{K}_i = \mathbf{T}_i (\sum \mathbf{W}_{son} - m_i \tilde{\mathbf{c}}_i) \qquad 34.$$

$$\mathbf{M}_i = (m_i \tilde{\mathbf{c}}_i)\mathbf{K}_i - m_i \mathbf{I} + \sum (\mathbf{Q}_{son}(\mathbf{I} - \tilde{\mathbf{l}}_i \mathbf{K}_i)) \qquad 35.$$

Along the way, we can accumulate some torque and force information for each segment, $\tau_{\sigma,part}$ accumulates torques, and f_σ accumulates forces. Note I'm assuming that external torques ($\tau_{ext,i}$) are being defined in terms of the local frame (and include torques due to external forces), but external forces ($_{ext,i}$) are in terms of the world space frame.

$$\tau_{\sigma,part,i} = -\omega_i \times (\mathbf{J}_i \times \omega_i) + \tau^i_{ext,i} + (m_i \mathbf{c}_i) \times \mathbf{R}^{fromworld} \mathbf{a}^{world}_{grv,i} \qquad 36.$$

$$f_{\sigma,i} = -\omega_i \times (\omega_i \times (m_i \mathbf{c}_i)) + \mathbf{R}^{fromworld}(\mathbf{f}^{world}_{ext,i} + m_i \mathbf{a}^{world}_{grv,i}) \qquad 37.$$

The following equations are the fastband, and should be done each time through the dynamics loop loop.

$$\tau_{\sigma,i} = \tau_{\sigma,part,i} - \tau_{topar,i} + \sum (\mathbf{R}^{topar}_{son} \tau^{son}_{topar,son}) \qquad 38.$$

$$\mathbf{d}_i = \mathbf{T}_i(\tau_{\sigma,i} + \sum (\mathbf{l}_{son} \times (\mathbf{R}^{topar_{son}} \mathbf{f}'^{son}_{son} + \mathbf{Q}_{son}\mathbf{a}_{c,son}))) \qquad 39.$$

$$\mathbf{f}'_i = \mathbf{f}_{\sigma,i} + (m_i \mathbf{c}_i) \times \mathbf{d}_i + \sum (\mathbf{R}^{topar}_{son} \mathbf{f}'_{son} + \mathbf{Q}_{son}(\mathbf{a}_{c,son} - \mathbf{l}_{son} \times \mathbf{d}_i)) \qquad 40.$$

Outward Pass: This completes the world traversing the tree inward. Now we traverse the tree outward, again the work can be divided into a slow and fastband depending on whether the information should be updated each time. (Typically I don't differentiate the two.) First the important accelerations of the root segment, the only one capable of translating freely.

$$\mathbf{a}_{root} = -(\mathbf{M}_{root})^{-1}\mathbf{f}'_{root} \qquad 41.$$

$$\dot{\omega}_{root} = \mathbf{K}_{root}\mathbf{a}_{root} + \mathbf{d}_{root} \qquad 42.$$

For the rest of the segments on the way out to the leaves

$$\mathbf{a}_i = \mathbf{R}^{frompar}(\mathbf{a}_{c,i_{par}} + \mathbf{a}^{par}_{par} - \mathbf{l}^{par} \times \dot{\omega}^{par}_{par}) \qquad 43.$$

$$\dot{\omega}_i = \mathbf{K}_i \mathbf{a}_i + \mathbf{d}_i \qquad 44.$$

$$\mathbf{f}_{topar,i} = \mathbf{M}_i \mathbf{a}_i + \mathbf{f}'_i \qquad 45.$$

if needed to check the solution.

Integration: Now we can integrate to find the new positions and velocities. This again consists of a step that needs to be done each time period, and a step that can possibly be done less often.

This step is done each time period. δu signifies an angular change vector accumulating orientation changes. Remember that while these values are defined in terms of the local frame orientation, they are inertial, including motion not only at the joint to the parent but all motion of all ancestors back to the world. For each segment,

$$\omega_{new} = \omega_{old} + \delta t \, \dot{\omega} \qquad 46.$$

$$\delta u_{new} = \delta u_{old} + \delta t \, \omega \qquad 47.$$

For the root segment, we are also interested in its linear motion. The linear motion of the other segments (here relative to the worldspace frame orientation) can be calculated from their angular motion.

$$\mathbf{v}_{new}^{world} = \mathbf{v}_{old}^{world} + \delta t \, \mathbf{R}^{toworld} \mathbf{a}_{new} \qquad 48.$$

$$\mathbf{p}_{new}^{world} = \mathbf{p}_{old}^{world} + \delta t \, \mathbf{v}_{new} \qquad 49.$$

Finally, we can update the rotation matrices at the slowband rate from distal to proximal (leaves to root). (Reset $\delta\mathbf{u}$ to zero after this operation.)

$$\mathbf{R}_{new}^{topar} = \mathbf{R}_{old}^{topar}(\mathbf{I} + \delta\mathbf{u}) \qquad 50.$$

This matrix should be orthonormalized to reduce error accumulation.[8]

Finally, each $\mathbf{R}^{topar}$ and its inverse can be calculated

$$\mathbf{R}_{new,son}^{topar} = \mathbf{R}_{new,i}^{fromworld} \mathbf{R}_{new,son}^{toworld} \qquad 51.$$

Armstrong and Green[3] suggest that the numerical instability that sometimes accumulates and causes bodies to flail about can be reduced by reducing the time step δt or by increasing artificially the moments of inertia about longitudinal axes. This latter method may produce some anomalous behavior, however, in my experience.

3.3.1. Control Issues

It is not terribly difficult to write subroutines to do the dynamics explained above (or to borrow the code from a friendly spirit who has done it before you). The open questions involve how to use this dynamic ability to get desirable motion and simulate constraints nicely. Some hints to solving these problems are presented in this section, but a great deal of work remains to be done before we can watch simulated animals moving realistically about on our computer screens under total dynamic control.

Clearly, the way to control the motion is to supply forces and torques that cause or restrict motion, either directly or through sophisticated preprocessors. Control could also be supplied in the form of extra constraint equations that limit the degrees of freedom involved. This method will not be discussed here.

3.4. Automatically Obvious : Gravity

The effect of gravity is easily calculated given the gravitational acceleration (about $9.81 m/sec^2$ on the earth's surface). Assuming the y-axis points away from the center of the earth, the force acting on the center of mass of each rigid body is

$$\mathbf{f}_{grv} = (0, -9.81, 0)\, m \qquad 28.$$

The torque due to this force acting in the body fixed coordinate frame is

$$\tau_{grv} = \mathbf{c} \times \mathbf{f}_{grv} \qquad 29.$$

3.5. External Dynamic Control

The user can shove the body about by applying forces and torques directly.

3.5.1. External Applied Torques

You can apply a pure *external* torque to cause rotation of the body about an axis by giving a 3d torque vector which is added to the net torque vector τ used in the dynamics equations for rotation.

3.5.2. External Applied Forces

Forces require both a 3d vector for the force itself and a 3d vector for its point of application. If is often most convenient to specify the force in terms of worldspace coordinates (converting it to the coordinates of the local frame of the segment upon which it is acting before doing the dynamics equations). The force itself is added to the net force used in the translational equations of motion **f**.

The position of the force is essential because the force may also cause a torque, depending upon where it is applied. It is usually most convenient to specify the torque in terms of the local coordinate frame, e.g., pick a local point of application **p**. The torque due to the force is found by Equation 16.

3.6. Internal Control

Internal control is mostly relevant to moving an articulated body in the way robots and animals move themselves, by applying torques and forces between neighboring segments. As the dynamics formulation described for articulated bodies only accommodates rotary joints, only internal torques, not forces will be mentioned.

3.6.1. Internal Torques

If you would like the torque to be *internal*, e.g., simulating a muscle that acts upon two neighboring segments in an equal and opposite fashion, this torque should contribute to the net torque on one segment and its negative should contribute to the net torque on its neighbor.

Internal torques are also useful for simulating joint limits, e.g., to keep the arm from bending backwards at the elbow. Rotary spring and damper combinations or exponential torques can be used to simulate them.

3.6.2. Positional Suggestions

Moving bodies about by suggesting forces and torques is less than intuitive. We usually think about motion kinematically, as changes in position. It is still possible to take advantage of dynamics but have the user think in positional terms by providing a (more or less) intelligent preprocessing step that converts positional suggestions to forces and torques that will accomplish them.

3.6.3. Internal Positional Control

The user could suggest local positional changes at joints, e.g., rotate the elbow from 45 degrees to 60 degrees in 10 seconds. The system could take into account the mass of the segments moving and their present velocity and guess how much internal torque will do this. Using super- or adaptive sampling or feedback, reasonable torques can be found to accomplish the desired motion. Before you ask why use dynamics at all, consider that only a few joints of the body need be under positional control at any time. The rest may be left in a simple state that is automatically dealt with, e.g., relaxed and hanging loosely, or frozen into a local configuration.

3.6.4. External Positional Control: Goals

It is sometimes handy to pick a point on a body and then a point in worldspace where you would like that point to be (a goal). In this case, you can apply a force starting at the desired body point and directed toward the goal. Finding the amount of force to pull the body to the goal at a reasonable speed without overshooting it or oscillating is sometimes tricky.

3.7. Environment Interactions

It would be nice if bodies could react automatically and realistically to their environment as well. This will add to the cost of the system, because considerable collision detection may have to be done. A simple brute force method of finding collisions is to check for the intersection of all the bounding vertices of an object with the bounding planes of all other objects.

3.7.1. Floors

Floors can be simulated with reasonable success by modeling them as a combination of a spring and a damper. A spring supplies a force dependent upon the amount its compressed, δc, times a constant k.

$$f_{spr} = k \delta c \tag{30.}$$

Similarly, a damper supplies a force dependent upon its velocity times a constant.

For complex articulated bodies, it may be well not to use a constant constant for these equations, but find some way of automatically calculating a reasonable proportionality constant for the body considering its total motion.

3.7.2. Other Collisions

Collisions with other objects is not fundamentally different, though their shapes may be different and they may be expected to move in response as well. In this case, the collision should be recognized and the collisions forces found before dynamics is done on the individual objects to find their motion in response to the collisions. For simple bodies, one might prefer to calculate the effects of collisions directly, rather than simulating them with springs and dampers.

4. Numerical Issues

Dynamics is alot more expensive than kinematics, but not unreasonably so, given the rapidly decreasing cost of compute power. I imagine we could be doing this on modern personal computers without too much trouble; at least, if I had a modern personal computer, I'd try it. The bells and whistles are costly, e.g. collision detection, joint limits, internal preprocessed control, etc. Lots of work remains to be done on this. Use of recursive dynamic formulations is a real boon. More sophisticated numerical integration methods can also help, Runge-Kutte integration is somewhat more complex to program and takes longer per time step but you can use much larger time steps than with the Euler method and get more accurate results. Adaptive calculations can also help, e.g. use large time steps when the body is falling freely but very small ones when it hits the floor. A clever adaptive idea (thanks to Ralph Abraham, UCSC) is to do a 5th order and a 4th order Runge-Kutte integration and if they deviate more than some allowed amount, redo it with a smaller time step.

5. Who is Doing It?

This is by no means a complete list, but the people and places that I have heard are doing this sort of thing include: me (at UC Santa Cruz), Dave Forsey (at the University of Waterloo),[24] Bill Armstrong and Mark Green (at the University of Alberta),[1, 3, 2] Michael Girard and A. Maciejewski[9] (at the Ohio State University). Work being presented at SIGGRAPH '87 relating to this topic includes that of Haumann at Ohio State[11] Al Barr and others (CalTech and elsewhere),[4] and Terzopoulos et al,[20] Isaacs and Cohen[13] and Witkin et al.[27]

6. Summary

This paper is a summary of the knowledge of dynamics that I've found useful for simulating the motion of bodies for computer animation. It's been an interesting and enjoyable way of creating animation, and seems to have a future. I hope you have fun with it and tell me if you have any problems or come up with any new solutions. Good luck!

Table 1. Meaning of Terms

Matrices	
$\mathbf{J}$	= *inertial tensor matrix*
$\mathbf{R}^{topar}$	= *rotation matrix segment to parent*
$\mathbf{R}^{frompar}$	= *rotation matrix parent to segment*
$\mathbf{R}^{toworld}$	= *rotation matrix segment to world*
$\mathbf{R}^{fromworld}$	= *rotation matrix world to segment*
$\mathbf{D}$	= *rotation matrix seen as direction cosines*
$\mathbf{I}$	= *identity matrix*
$\mathbf{K}$	= *recursive coefficient matrix*
$\mathbf{M}$	= *recursive coefficient matrix*
3d Vectors	
$\mathbf{f}$	= *force*
$\mathbf{f}_{grv}$	= *force due to gravity*
$\mathbf{f}_{ext}$	= *external applied force*
$\mathbf{f}_{son}$	= *force applied by child of a segment thru a joint*
$\mathbf{f}_{topar}$	= *force applied onto parent of a segment thru a joint*
τ	= *torque*
τ_{grv}	= *torque due to gravity*
τ_{ext}	= *external applied torque*
τ_{son}	= *torque applied by child of a segment thru a joint*
τ_{topar}	= *torque applied onto parent of a segment thru a joint*
$\mathbf{p}$	= *position*
$\mathbf{v}$	= *linear velocity*
$\mathbf{a}$	= *linear acceleration*
$\mathbf{a}_{grv}$	= *gravitational acceleration*
$\delta\mathbf{u}$	= *angular position*
ω	= *angular velocity*
$\dot{\omega}$	= *angular acceleration*
$\mathbf{l}$	= *vector to joint of son segment from parent frame*
$\mathbf{c}$	= *vector to segment center of mass defined in segment frame*
$\mathbf{d}$	= *recursive coefficient*
$\mathbf{f}'$	= *recursive coefficient*
Scalars	
m	= *mass*
δt	= *time step between samples*
I_x, I_y, I_z	= *moments of inertia*
I_{xy}, I_{xz}, I_{yz}	= *products of inertia*

Appendix I. Runge-Kutta Methods

by Matthew Moore
UCSC Santa Cruz

The Euler method of numerical integration is not very accurate; the Runge-Kutta methods are superior and worth the slight effort of writing more complex code. The Runge-Kutta methods have been described as the workhorses for solving differential equations. Several other methods are available, such as the predictor-corrector family, which provide incremental speed advantages over Runge-Kutta and which are less sensitive to stiffness in the equations.

As an example, here is a 4-th order Runge-Kutta method for producing a numerical solution to the 2'nd order ordinary differential equation

$$\frac{d^2P}{dt^2}=\frac{F(P,t)}{m}$$

which defines the motion of a point mass under an applied force. First decompose into two 1'st order ODEs.

$$\frac{dV}{dt}=F(P,t)$$

$$\frac{dP}{dt}=V$$

Now step through the time interval of interest generating a sequence of positions for the point. The step size is *h*. This set of equations generates the *n+1'st* position and velocity vectors, given the *n'th*.

$$Kp_1=hV_n$$

$$Kv_1=h\frac{F(P_n,t)}{m}$$

$$Kp_2=h(V_n+\frac{Kv_1}{2})$$

$$Kv_2=h\frac{F(P_n+\frac{Kp_1}{2},t+\frac{h}{2})}{m}$$

$$Kp_3=h(V_n+\frac{Kv_2}{2})$$

$$Kv_3=h\frac{F(P_n+\frac{Kp_2}{2},t+\frac{h}{2})}{m}$$

$$Kp_4=h(V_n+Kv_3)$$

$$Kv_4=h\frac{F(P_n+Kp_3,t+h)}{m}$$

$$P_{n+1}=P_n+\frac{Kp_1+2Kp_2+2Kp_3+Kp_4}{6}$$

$$V_{n+1}=V_n+\frac{Kv_1+2Kv_2+2Kv_3+Kv_4}{6}$$

This is already a system of 6 first order ODEs. The extension of this algorithm to larger numbers of point masses is straightforward.

Adaptive Stepsize Control

In solving differential equations, a phenomenon called stiffness is often encountered. Stiffness causes numerical instability if the integration step size is too large. Choosing a step size which will cope with the maximum stiffness which could ever be encountered would unjustly penalize the algorithm during more normal cases. It is best to include a facility to adaptively determine the step size that is appropriate as the dynamics algorithm runs.

For example, the adaptive stepsize algorithm could work by solving the dynamics problem twice in each time step, using both a 4'th order Runge-Kutta and a 5'th order Runge-Kutta for this. If the two solutions are not within a given tolerance of each other at every point in the model, the algorithm would backtrack to the beginning of the time step and halve step size. If the two algorithms agree with each other to within a second, much smaller, tolerance at every point, then the step size can be doubled for the next step. This requires over twice as much work as a single 4'th order Runge-Kutta solution for each step, but the investment is handsomely repaid when the algorithm takes large steps through non-stiff situations.

Intuitively, stiffness arises in a system with an equilibrium configuration, and a restoring force which pushes the system back towards that configuration whenever the system moves away from it. An example of this would be a feedback controller which exerts a torque on a joint with the intention of maintaining some given joint angle. In numerical integration with a finite step size, it is possible for the step to be too large, so that the system is taken from near the equilibrium to some other configuration on the "other side" of the equilibrium and farther away. Thus the integration "overshoots" the equilibrium configuration. The next step will see the restoring force even stronger, and will overshoot even more. The system may then oscillate farther and farther away from the equilibrium configuration without bound, until completely ridiculous values are produced. The solution is to reduce the step size until overshoot does not occur. As the restoring force becomes stronger, overshooting becomes easier, and so the step size must be reduced even further.

References

1. William W. Armstrong, ''Recursive Solution to the Equations of Motion of an N-link Manipulator,'' *Proceedings Fifth World Congress on the Theory of Machines and Mechanisms*, pp. 1343-1346, Am. Soc. of Mech. Eng., 1979.
2. William W. Armstrong, Mark Green, and R. Lake, *Proceedings of Graphics Interface 86*, pp. 147-151, May, 1986.
3. William W. Armstrong and Mark W. Green, ''The Dynamics of Articulated Rigid Bodies for Purposes of Animation,'' *Proceedings of Graphics Interface '85*, pp. 407-415, Computer Graphics Society, May, 1985.
4. Alan H. Barr, ''Dynamic Constraints,'' *SIGGRAPH '87 Tutorial Notes: Topics in Physically-Based Modeling*, 1987.
5. S. Conte and C. de Boor, in *Elementary Numerical Analysis, 3rd edition*, McGraw-Hill Book Company, New York, 1980.
6. R. Featherstone, ''The Calculation of Robot Dynamics Using Articulated-Body Inertias,'' *International Journal of Robotics Research*, vol. 2, no. 1, pp. 13-30, Spring, 1983.
7. Richard P. Feynman, Robert B. Leighton, and Matthew Sands, in *The Feynman Lectures on Physics*, California Institute of Technology, Pasadena, California, 1963 .
8. Daniel T. Finkbeiner, II, *Introduction to Matrices and Linear Transformations*, p. 174, W. H. Freeman and Company, San Francisco, CA, 1960. matrices
9. Michael Girard and Antony A. Maciejewski, ''Computational Modeling for the Computer Animation of Legged Figures,'' *SIGGRAPH '85 Conference Proceedings*, vol. 19, pp. 263-270, July, 1985.
10. Donald T. Greenwood, in *Principles of Dynamics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.
11. David Haumann, ''Modeling Flexible Bodies,'' *SIGGRAPH 1987 Tutorial Notes: Topics in Physically-Based Modelling*, July, 1987.
12. Roberto Horowitz, ''Model Reference Adaptive Control of Mechanical Manipulators,'' PhD Thesis, Mechanical Engineering, University of California, Berkeley, California, May, 1983.
13. Paul M. Isaacs and Michael F. Cohen, ''Controlling Dynamic Simulation with Kinematic Constraints,'' *SIGGRAPH 1987*, July, 1987.
14. C. S. George Lee, R. C. Gonzalez, and K. S. Fu, *Tutorial on Robotics*, IEEE Computer Society Press, Silver Spring, MD, 1983.
15. W. G. McLean and E. W. Nelson, *Engineering Mechanics: Statics and Dynamics*, Shaum's Outline Series, McGraw-Hill Book Co., New York, 1978.
16. NASA, *Anthropometric Source Book*, NASA Scientific and Technical Information Office, 1978.
17. L.A. Pars, *A Treatise on Analytical Dynamics*, Ox Bow Press, Woodbridge, Connecticut, 1979.
18. Richard P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*, The MIT Press, Cambridge, MA, 1981.

19. Robert Resnick and David Halliday, *Physics Part I,* John Wiley and Sons, Inc., New York, 1966.

20. Demetri Terzopoulous, John Platt, Alan H. Barr, and Kurt Fleischer, ''Elastically Deformable Models,'' *SIGGRAPH 1987*, July, 1987.

21. Dare A. Wells, *Lagrangian Dynamics,* Shaum's Outline Series, McGraw-Hill Book Co., New York, 1969.

22. Jane Wilhelms, ''Virya - A Motion Control Editor for Kinematic and Dynamic Animation,'' *Proceedings of Graphics Interface 86*, pp. 141-146, May, 1986.

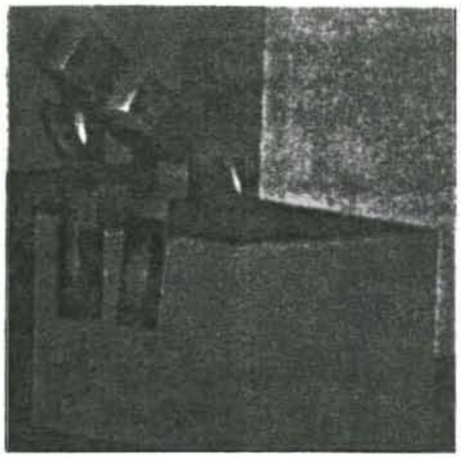
23. Jane Wilhelms, ''Using Dynamic Analysis for Animation of Articulated Bodies,'' *IEEE Computer Graphics and Applications*, vol. 7, no. 6, June, 1987.

24. Jane Wilhelms, David Forsey, and Pat Hanrahan, *Manikin: Dynamic Analysis for Articulated Body Manipulation,* Computer and Information Sciences Board, U. of California, Santa Cruz, CA 95064, April, 1987. Tech. Report UCSC-CRL-87-2

25. Jane Wilhelms, ''Graphical Simulation of the Motion of Articulated Bodies such as Humans and Robots, with Particular Emphasis on the Use of Dynamic Analysis,'' *PhD Thesis*, Computer Science Division, Berkeley, CA, July, 1985.

26. Jane Wilhelms and Brian A. Barsky, ''Using Dynamic Analysis for the Animation of Articulated Bodies such as Humans and Robots,'' *Proceedings of Graphics Interface '85*, pp. 97-104, May 1985.

27. Andrew Witkin, Kurt Fleischer, and Alan H. Barr, ''Energy Constraints on Parameterized Models,'' *SIGGRAPH 1987*, July, 1987.



Toward Automatic Motion Control

Jane Wilhelms
University of California, Santa Cruz

Motion control for computer animation is a rich area for new research. The trend toward greater complexity in animation makes the development of more convenient and automatic methods of motion control important. Most commonly used motion control methods, such as keyframing and scripts, require a great deal of user effort to design acceptable animations. More automatic methods will allow the production of sophisticated animation with less user effort. These methods include dynamic analysis, path planning and collision avoidance, stimulus-response control, and learning algorithms.

The field of motion control for computer animation is still in its youth, though perhaps no longer in infancy. A wide variety of techniques are in use, many in combination. Designating motion for complex 3D scenes is complicated and time-consuming, because of both the difficulty in choosing from the wide range of motion possible, and the sheer amount of information that must be specified. Commonly used approaches are fairly low level and require a considerable amount of user input to design the motion. But as motion becomes more realistic, with many objects interacting in complicated ways, providing more automatic and high-level approaches to motion control becomes increasingly important. This article concentrates on present and potential approaches to automating motion control for computer animation.

Overview

Motion control systems are often classified as interactive, scripted, or a combination of both. Interactive implies that the user and motion control system participate in a loop where the user describes a motion, the computer quickly provides an animation using it, the user modifies the motion as necessary, etc. In scripted methods, the user creates a written script describing the motion that the control system later interprets to produce the animation.[1]

Motion control can also be classified as low level, high level, or somewhere in between. Low level suggests that the user is required to specifically describe motion for individual degrees of freedom, such as a path through space or motion at joints. With high-level control, the user may describe motion in more general terms, as in "walk forward," leaving the system to find the appropriate low-level motion description. High-level control is desirable because the user can succinctly and quickly define complex motion. It does have some disadvantages, however, because the user has less control over the exact motion. For this reason, systems typically provide a combination of high-level and low-level techniques.

High-level motion control systems are difficult to develop because converting general descriptions to specific instructions is complicated. Systems may make the interpretation from high-level to low-level commands by relying on command libraries, finite state machines, hierarchies,[2] and parameterization.[1,3,4] These methods still initially require considerable user motion specification to define the libraries, state machines, types of parameters, etc. Furthermore, if the objects in the modeled environment change, the system may have to be instructed how to respond to the new environment.

The intelligence of the control system can be increased in a number of ways, reducing the amount of user input required. Drawing upon methods developed in robotics, such as inverse kinematics and path planning, moving objects can be constrained to move in a desirable manner in any environment. Using algorithmic control, motion can be generated through precise steps. In an extremely intelligent system, objects could generate their motion on the fly by responding to a changing environment and even learn from their experiences. Use of stochastic processes might allow introduction of (pseudo-) randomness into the motion, making it more interesting and possibly more realistic, though there is danger of this leading to jittery and unnatural movement.

Interactive methods

Interactive refers to motion control techniques that allow the user to design motion in real time while watching the animation develop on the graphics screen. The user can take full advantage of the dials, joysticks, tablet, and other interactive devices to quickly modify the motion and see the result.

The intelligence of the control system can be increased in a number of ways, reducing the amount of user input required.

The earliest, and still one of the most common, interactive motion control methods is keyframing. Here, the user specifies a sequence of positions and the times when they occur, and the computer interpolates between these positions to produce the animation. The interpolation generally uses some splining technique to provide at least first derivative (velocity) and sometimes second derivative (acceleration) continuity to the motion. 2D keyframing has certain serious limitations,[5,6] but 3D object-based keyframing is a convenient and successful motion control method.[7] Keyframing has the advantage that the user can see the total configuration of the system at given times, easily noting collisions or undesirable interactions. The disadvantages are that it is low level, requires the user to specifically control each degree of freedom, and does not allow easy visualization of the motion between keyframes.

Another interactive motion control method is path specification. While keyframing involves positioning all the degrees of freedom of the system for particular time points, path specification involves designating a coherent path over time.[8,9] Often this path is interactively created by using dials or a tablet to pick positions in a displayed world. These positions can then be used to define smooth, curved paths along which objects move during the animation.

The advantage to path specification is that the user can think in terms of the entire motion of a particular body. On the other hand, it is low level and does not allow easy visualization of the configuration of all objects at a particular time instant (the inverse of the keyframing problem).

A third interactive motion control method is the use of control functions,[10] where motion is specified individually for each degree of freedom as a function of time versus position. Again the functions are generally developed by designating control points which define curves. Control functions have the advantage that motion described by other control methods can be easily stored in this form, and specific low-level changes are easily made to individual degrees of freedom. However, control functions are less easily visualized as motion in the animated world than world space keyframes or paths and are very low level.

Scripted animation

Scripted animation (or animation languages) allows motion to be described as a formal, written script.[11,12] Such a script can be unambiguously interpreted by the animation system, which can carry out all later stages of the animation independent of further user input. Some languages, such as GRAMPS,[13] are fairly low level. GRAMPS combines scripted and interactive animation, and the script facilitates communication with the device, a calligraphic display. More high-level languages, such as ASAS,[11] allow a wider range of object control, including independence or interaction of objects, synchronization, and ordered sequencing of events over time. ASAS is a complete programming language based on Lisp and includes such structured programming options as procedures, recursion, and typed data structures, as well as uniquely graphical objects and operators.

Scripted animation systems for articulated body movement have been developed which take advantage of dance notations.[14] Both Benesh notation[15,16] and Labanotation[17-19] have been used to describe motion for computer graphics.

Motion control languages are also of interest in robotics.[20] Languages provide the ability to communicate with the machine in high-level constructs, to describe a complete task, and to integrate sensory feedback into the control system. They provide the potential for high-level decision-making, and may be classed as explicit or implicit. Explicit languages are low level; the user must describe the motion in terms of specific position changes and velocities. Implicit languages are more high level; the user describes the task to be accomplished, relying upon an intelligent system to find how to complete it. Examples of robot languages are WAVE, AL, and AUTOPASS.[21-23]

High-level to low-level interpretation

Given that the user does not always want to specify motion at the lowest possible level (that is, as numbers controlling degrees of freedom), various means have been developed to provide the user with higher level control. These include parameterization, finite state machines, command libraries, and hierarchies.

Parametric motion control involves designating parameters whose values define the configuration or motion of the objects modeled.[1,3] For example, in the case of facial animation, parameters may be used to alter the position of the mouth or the elevation of the eyebrows.[3] Parameters are convenient and allow association of reasonably complex motions (such as smiling) with relatively few commands. While parameterization can be easily implemented using procedures, choosing parameters that cover the desired range of motion can be problematic. The user may have to sacrifice fine control for ease of use.

Finite state machines are appropriate for describing and controlling repetitive or coordinated motion.[2,24] For example, Zeltzer uses a finite state machine with four states to control the walking of articulated figures. The four states of the legs are left stance/right swing, left stance/right stance, left swing/right stance, and left stance/right stance.

Command libraries provide a means of storing low-level motion descriptions under high-level command names. The command "walk," for example, may stand for complex changes to many joints of an articulated body. Command libraries are a convenient, adaptable way to implement high-level control. Combining commands can present problems, however, because commands may send different directions to the same joints and produce nonsensical motion.[17]

A hierarchical structure that uses levels of progressively more detailed, less general directions to interpret high-level commands may make combining commands less problematic and reduce the number that need to be stored. For example, if the commands for walking are treated as a combination of commands for whole limbs, which are in turn treated as commands to individual joints, the commands to raise the leg could be used with a variety of other motions besides walking. Zeltzer's skeleton animation system combines the use of command libraries and a hierarchical interpreter with finite state machines.[2]

Possibilities of more automatic control

The methods described above require a considerable amount of user input to design motion. Even in the case of high-level control by parameterization and command libraries, the user or programmer must at some point describe the low-level position changes that correspond to "walk" or "smile." As the environment and the desired motion increase in complexity, it may be desirable to rely upon a more intelligent, versatile, and automatic form of motion control. An intelligent control system should take advantage of the knowledge available in other fields, such as robotics, artificial intelligence, and physics.

Motion control for computer animation is very similar to that of mechanical manipulators. Therefore, it is not surprising that robotics has much to offer this field. For example, robot control must consider the dynamics of the situation: How will the masses of the manipulator respond to environmental forces and torques? Most computer animation describes motion strictly kinematically, thinking only in terms of positions versus time. The use of dynamics can help add realism to motion.[25]

In robotics, the interest lies in inverse kinematics (how to position a jointed body so that its distal end is at a particular position in space), and path planning and collision detection (how can the robot find a path through space when obstacles are present). All these issues are equally pertinent to animation control. Indeed, the present state of robot control is not unlike the present state of animation control. Most industrial robots use "teaching by doing" for control, where the robot is led through positions which it then emulates.[20] This is much the same as the way objects are controlled in computer animation using keyframing or path specification. Some robots can be controlled by robot languages, similar to animation languages.[26]

Other interesting approaches might be described as algorithmic control, stochastic control, stimulus-response behavioral control, and learning. Algorithmic control refers to motion developed through the use of a particular algorithm, as in modeling tree growth by algorithmically defining rate of expansion, branching pattern, and leaf production. Stochastic control implies a degree of controlled randomness, usually added to motion defined by other methods. Stimulus-response control is meant to suggest that object motion is determined not only by some predetermined pattern, but by responding to stimuli encountered during the motion. For example, a pedestrian moving through a city might respond to oncoming traffic by moving to the sidewalk. Learning refers to changing the motion dependent upon experience. In this case, the pedestrian might have to learn to move to the sidewalk by being consistently run over by cars.

Types of objects and their motion

Objects vary in the types of motion available to them (see Figure 1). Typically motion is described in terms of degrees of freedom, that is, the number of independent coordinates needed to specify the positions of all components of the system. For a system with n degrees of freedom shown in t animation frames, a total of $n \times t$ numbers must be specified. A realistic example is a video animated at 30 frames a second. A three-minute animation of a set of objects with 50 total degrees of freedom (not an unreasonably complex situation) requires 4500 numbers to completely specify the motion. Of course, use of keyframing and spline techniques reduces the percentage of these numbers that the user must specify, but the amount of data needed for motion control is still considerable.

Particles

A particle can be described as a point in three space (x, y, z). The location and motion of such a point is designated by three variables, and thus the point has three degrees of freedom of motion. Animating a point requires a triplet of numbers for each animation frame, or three functions describing the variation of x, y, and z over time. Reeves et al. have shown how useful particles can be in simulating natural phenomena.[27,28]

Rigid bodies

A rigid body is defined by some number of points that must move together. They may not move relative to each other, though they may move as a whole relative to the world space. These points may define polygons or a free-form surface. The motion of a rigid body is specified by six degrees of freedom: x, y, z translation (as for a point) and x, y, z rotation (orientation). (Use of a Cartesian system is not necessary, though it is the most familiar method.) Motion of a rigid body is usually visualized as motion of a frame that is fixed to the rigid body, with all points defining the body moving with the frame. Most animation systems concentrate on modeling with rigid bodies.

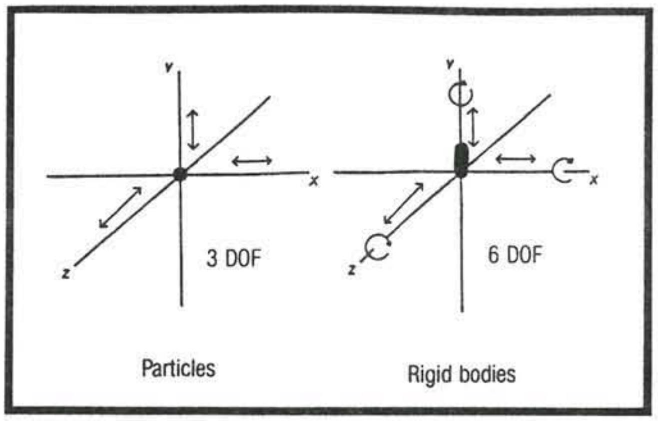
Flexible bodies

A flexible body consists of an infinite number of points which move relative to each other over time. In practice, a flexible body may be defined as a set of moving points. These points may represent vertices of polygons, but greater flexibility is achieved if they represent control points for surfaces. A moving flexible body, such as an amoeba, defined as a free form surface using p control points, has $3 \times p$ degrees of freedom varying over time. This explains why most computer animation is of rigid bodies. While some work has been done in the field,[29] animating flexible bodies remains a fertile area for research.

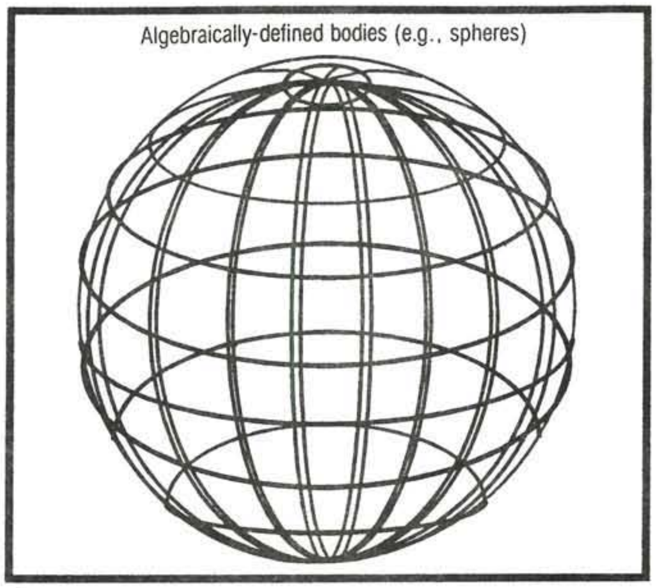
Articulated bodies

Articulated bodies are made up of segments whose motion relative to each other is somewhat restricted. For example, a human body is often represented as rigid segments joined at articulations (joints) which have one to three degrees of freedom. The total number of degrees of freedom that must be specified is the sum of the number of degrees of freedom at each joint. A good deal of work has been done modeling articulated bodies as rigid segments.[2,17,30-32]

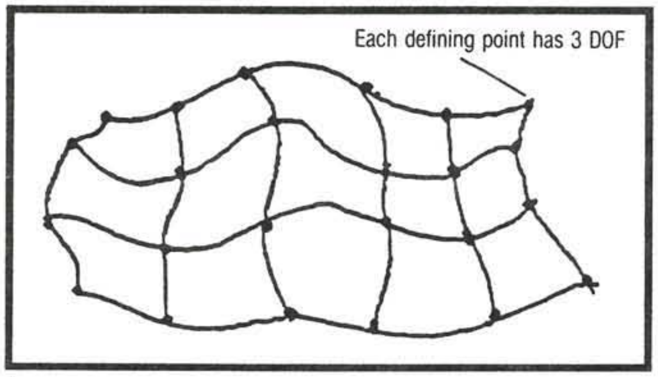
Articulated bodies may also be flexible. In this case, the joint between two flexible segments could be modeled as a joint between two coordinate frames, one attached to each adjacent segment. Two types of motion are then possible. Motion at the articulation consists of changing up to six numbers specifying the relation between frames. Motion within each flexible segment consists of moving the points defining the segment relative to its local frame. The total number of degrees of freedom necessary to specify a flexible articulated body would be the number of degrees of freedom at each joint plus three times the number of defining points.



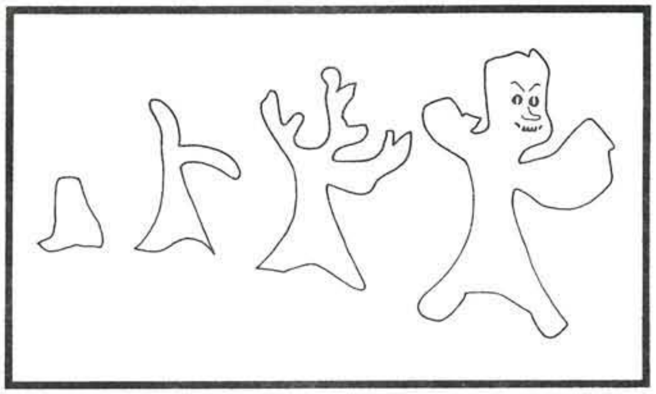
a



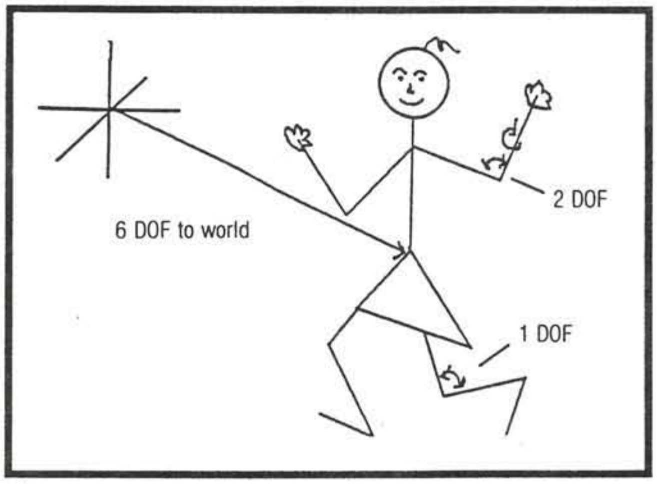
d



b



e



c

Figure 1. Degrees of freedom of motion: (a) particles and rigid bodies, (b) flexible bodies, (c) articulated bodies, (d) miscellaneous bodies, (e) metamorphosing bodies.

Miscellaneous and metamorphosing bodies

Bodies may also be specified algebraically. A sphere can be modeled as a center (three numbers) and a radius, and a pulsating sphere could be animated by changing the radius. The number of degrees of freedom of the system would depend upon the nature of the equations defining these algebraic objects.

It becomes more complex when the bodies being modeled change their number of degrees of freedom over time. Examples of metamorphosing bodies include fractal mountain ranges during formation,[33] growing plants,[34,35] and particle systems.[27] The process of metamorphosis is in itself a kind of motion control and will be dealt with below.

Constraints

Constraints restrict object motion (see Figure 2), and some entirely eliminate a degree of freedom. For example, the hinge joint at the knee may be modeled as constrained to move using only one revolute degree of freedom. Constraints can also be applied as limits within a degree of freedom. The head can swivel on the neck but cannot rotate all the way around.

These partial constraints can be difficult to model, because the limits on a degree of freedom may vary according to the positions at other degrees of freedom. For example, the arm cannot swing toward the body center when the body is upright, because the trunk is in the way. However, if the trunk is bent forward at the waist, such a motion is possible.[36]

Constraints may only be applied occasionally, as in the motion of leaping, where the legs are sometimes constrained by the ground. Constraints may take many forms: A robot hand may be constrained to remain in contact with a surface, or to move around a point. They can also be applied to velocity and acceleration, and the nature of the constraints may vary with time.

A motion control system should take into account constraints, or at least do constraint-checking and respond reasonably when they are broken. A reasonable response might include informing the user of the constraint violation, and possibly include refusal to accomplish the motion in violation.

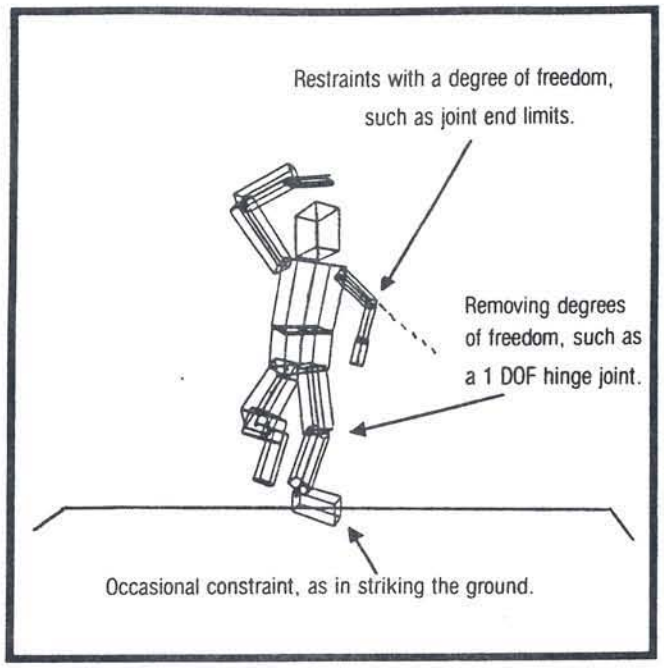

Restraints with a degree of freedom, such as joint end limits.
Removing degrees of freedom, such as a 1 DOF hinge joint.
Occasional constraint, as in striking the ground.

Figure 2. Constraints on motion.

Approaches to automating animation

More automated motion control will take the burden of motion specification away from the user and give it to the animation system. Ideally, one could imagine an automated system that, given a description of an environment, its movable bodies, some behavioral rules, and (perhaps) what kind of motion is desired, will create an appropriate and attractive animation. Such a system would require a more sophisticated program and probably be much more computationally expensive than user-directed systems. Furthermore, this type of system may not be desirable because it will take control of the exact motion away from the user. It is, however, possible for the user to alter such an animation at a lower level.

Such a system could also provide a useful tool for studying simulations. A simulation models behavior on the basis of known or hypothetical physical laws. Not all simulations are graphical, but many simulations can be better understood if combined with graphical output. Computer-generated graphical simulations are a kind of computer animation. Because such simulations try to imitate physical processes, they are naturally constrained to be realistic. Simulation techniques can be very useful in adding realism to computer animations.

Dynamic analysis

Typically, motion control is kinematic, with motion being specified by designating positions taken over time for each degree of freedom. Kinematic motion specification does not take into account the causes of motion, which are the effects of forces and torques acting upon masses. This is the concern of dynamics, or dynamic analysis. (Dynamics also refers to motion or change in the general sense. In this article, its use will be restricted to the meaning of the word as used in physics.)

Dynamics has been used in computer modeling in a few cases. Some CAD/CAM systems include engineering packages for dynamic analysis of machines. Vehicle crash studies may include dynamic simulations involving computer graphics.[37] The famous *Works* mechanical ants and robots from the New York Institute of Technology included some dynamic analysis.[38] The walking creatures modeled by Girard and Maciejewski[39] include simple dynamics of the trunk. In addition, I have explored,[32] and Armstrong and Green[40] have also independently modeled, articulated bodies using a more complete version of dynamic analysis.

The dynamics equations of motion are used to relate the acceleration of the mass (object) to the forces and/or torques acting upon it. A force produces translational motion; a torque produces revolute motion. In the simplest case, for a point mass capable of three degrees of freedom, the common Newtonian equation of motion is

$$\boldsymbol{F} = m\,\boldsymbol{a}$$

where $\boldsymbol{F}$ is the force vector acting on the point mass m and $\boldsymbol{a}$ is the acceleration the mass experiences. For constant mass, this equation can be solved in either of two directions: (1) given the force, one can solve for acceleration; or (2) given the acceleration, one can solve for force. In robotics, the second direction is generally of interest, as the desired motion is known and the amount of force that must be applied to the robot motors to pro-

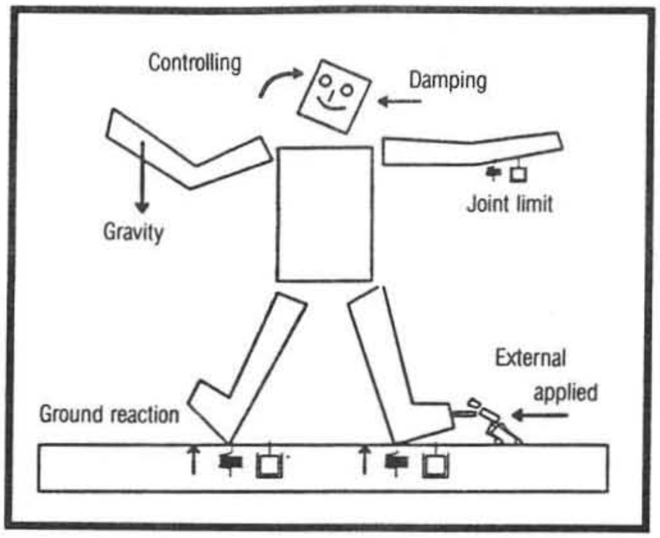


Figure 3. Forces and torques acting on the body.

duce this motion must be determined. In computer graphics, the first direction is of more interest: Given the environmental and internal forces and torques, what is the acceleration the object will undergo? Given the acceleration, present position, and velocity, the path of motion can be found. For simple systems with few degrees of freedom, analytical integration can be used. As the complexity of the system grows, numerical integration techniques are relied upon.

The dynamics equations for articulated bodies are complex. For greater realism, body segments should be treated as extended masses. Because of the interaction between connected segments, the dynamics equations are coupled and must be solved as a system of equations, one equation for each degree of freedom. There are a variety of formulations for the dynamics equations that produce the same result by different methodology. Two methods used in computer graphics are the matrix formulation of the Gibbs-Appell Equation[25] and a recursive formulation developed by Armstrong.[40] While each has advantages and disadvantages, the latter is much faster ($O(n)$ compared to $O(n^4)$ for the Gibbs-Appell version) and so is likely to be the method of choice.

Of the forces and torques acting on and in the body, some can be calculated automatically, some can be simulated using springs and dampers, and some must be supplied by the user (see Figure 3). For example, the gravitational force can be calculated automatically. Interactions with the ground, other collisions, and joint limits can be modeled as springs and dampers. Such internally controlled motion as muscles in animals or motors in robots are normally supplied by the user. Thus, certain types of complex motion, such as falling, striking the floor, or bouncing against other objects, can be automatically calculated and realistically rendered with no user input.

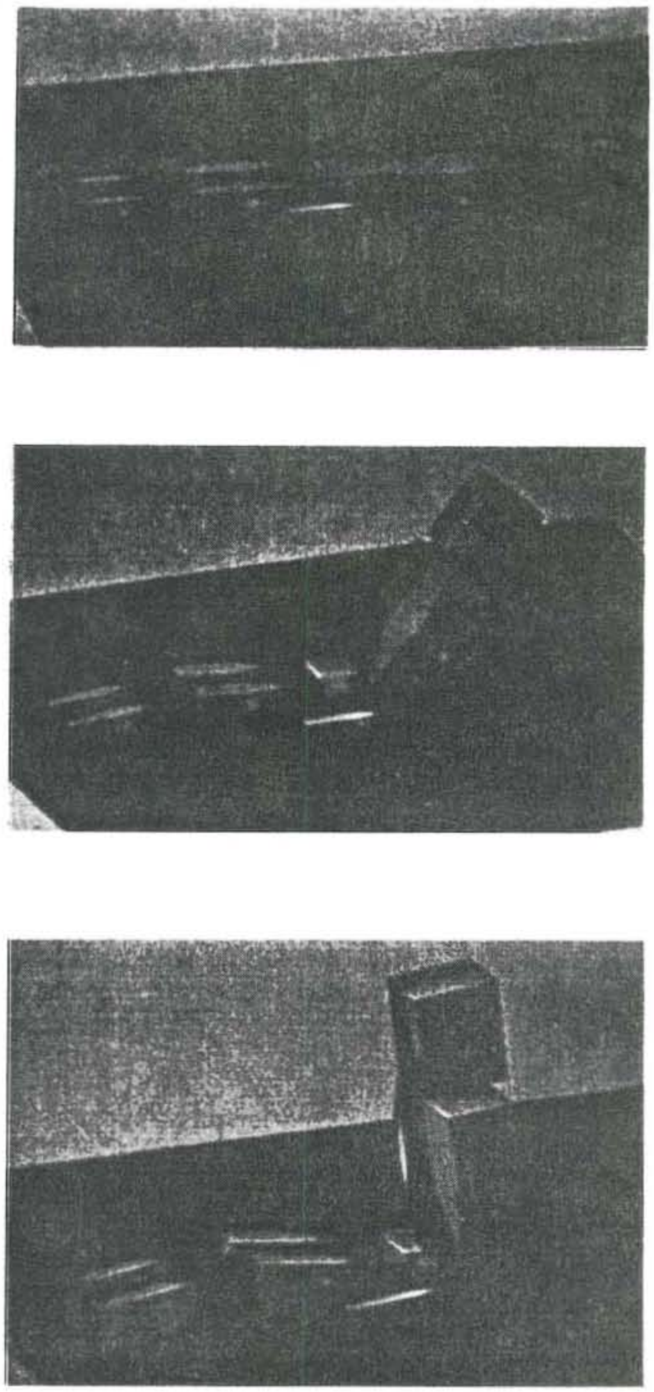


Figure 4. Dynamically controlled figure sitting up.

Producing controlled, coordinated motion using dynamic analysis is more of a problem. The user can provide control functions to specify pseudo-muscular forces and torques acting on individual degrees of freedom, but force/torque control is nonintuitive. It is also possible for the user to suggest the motion at joints under internal control as kinematic motion changes, and have the program calculate appropriate forces and torques to accomplish the motion. Figure 4 shows a body made to sit up using dynamic analysis. A torque is applied to the waist, the neck and hips are held stable, and all other joints are relaxed. The body naturally conforms its motion to the floor and gravity. Various approaches to controlling articulated bodies using dynamic analysis have been handled in detail in other studies.[10,25]

It is possible to model dynamics more simply by treating the body segments as point masses or limiting dynamic analysis to a subset of body parts.[39] *Ad hoc* methods may also be added, such as approximating the bouncing expected upon collision and adding this to kinematically described motion as a last step. These methods are simpler than full dynamics analysis but may not be as effective.

Inverse kinematics

Kinematics of articulated bodies involves two related problems: the direct kinematics problem and the inverse kinematics problem. The direct kinematics problem, which can be solved rather trivially, is to find the world space position of a distal segment of the body given the joint positions of the segments proximal to it. In other words, given the angles of the shoulder, elbow, and wrist, find the position of the hand. The inverse kinematics problem is to find the joint positions of the proximal joints given the position in world space of a distal segment; or, given the position of the hand, find the joint angles at the wrist, elbow, and shoulder that will place the hand in that position. Inverse kinematics provides a means of constraining articulated bodies in reference to the world. For example, it can be used to keep a body realistically in contact with the ground when it moves. Because kinematic motion has no concept of gravity or reaction forces, modeling such environmental interactions is nontrivial.

Inverse kinematics is important in both computer graphics and robotics. It involves two related problems: (1) finding any solution that will achieve the desired goal and (2) finding the most desirable solution. The inverse kinematics problem becomes progressively more difficult as the number of degrees of freedom increase. For simple six degree-of-freedom industrial robots, analytic, iterative, and geometric solutions exist.[20,26] Even in this simple case, however, there is often a problem of choosing the best solution.

For a complex body such as a simulated human figure, the problem becomes extremely unwieldy. Consider, for example, the number of ways a seated figure might reach for an object on the table in front of it. While many solutions are possible, certain solutions are much more natural and realistic. What are the criteria that yield a realistic solution? Considerations might be total energy expended, distance traveled by the hand, distance traveled by all the joints, time taken, and a subjective criterion such as naturalness.

Korein[36] has devised a technique called the reach hierarchy method for inverse kinematics that involves precomputing the workspace for the chain of articulated segments and for each of its distal subchains. The workspace is the region in space that the end of the segment is capable of reaching. The algorithm works as follows for point goals: If the goal is not in the workspace of the entire chain, it is unreachable, so give up. Or else, for each of the joints in the chain from proximal to distal, adjust the joint positions just enough so that the next more distal workspace includes the goal. This method minimizes adjustment of proximal joints.

Girard and Maciejewski[39] presented a technique for solving the inverse kinematics of legged figures by linearizing the relation between changes in joint space and changes in world space. A Jacobian matrix expressing this relation about the current operating point is reasonably accurate for small deviations from the present positions. The pseudo-inverse of the Jacobian can be used to find joint positions that will result in the desired world space position of the segments of interest.

It should be noted that some of the necessity of inverse kinematics can be eliminated by using dynamic analysis. For example, if the dynamically controlled body strikes the floor, the floor will push back with a reaction force sufficient to keep the body from falling through, realistic bouncing results. Dynamic forces can also be used to pull segments to a certain point in space.

Path planning and collision detection

Path, or trajectory, planning involves describing a path for an object through space. The user may explicitly define this path, either by defining keyframe positions to be passed through along the way, or by actually defining a path through space.[7-9] However, using automatic path-planning algorithms, it is possible to merely designate the start and goal positions and have the system determine the path, taking into account the environment and constraints. For single, nonarticulated body motion, Cartesian coordinates are used to specify a sequence of points through or near which the body passes.

For articulated bodies, motion may be described either in Cartesian coordinates or in joint coordinates. Cartesian motion is often used to describe how a particular part of the body, such as the hand, moves in world space. Joint coordinate motion is used to describe the local joint positions during the motion. Cartesian coordinates are a more intuitive way to describe motion through a world, but may require the solution of the inverse kinematics problem. A reasonable compromise is to specify world space trajectory points along a Cartesian path, using inverse kinematics to find the joint coordinate positions to achieve these positions. Simple interpolation of joint

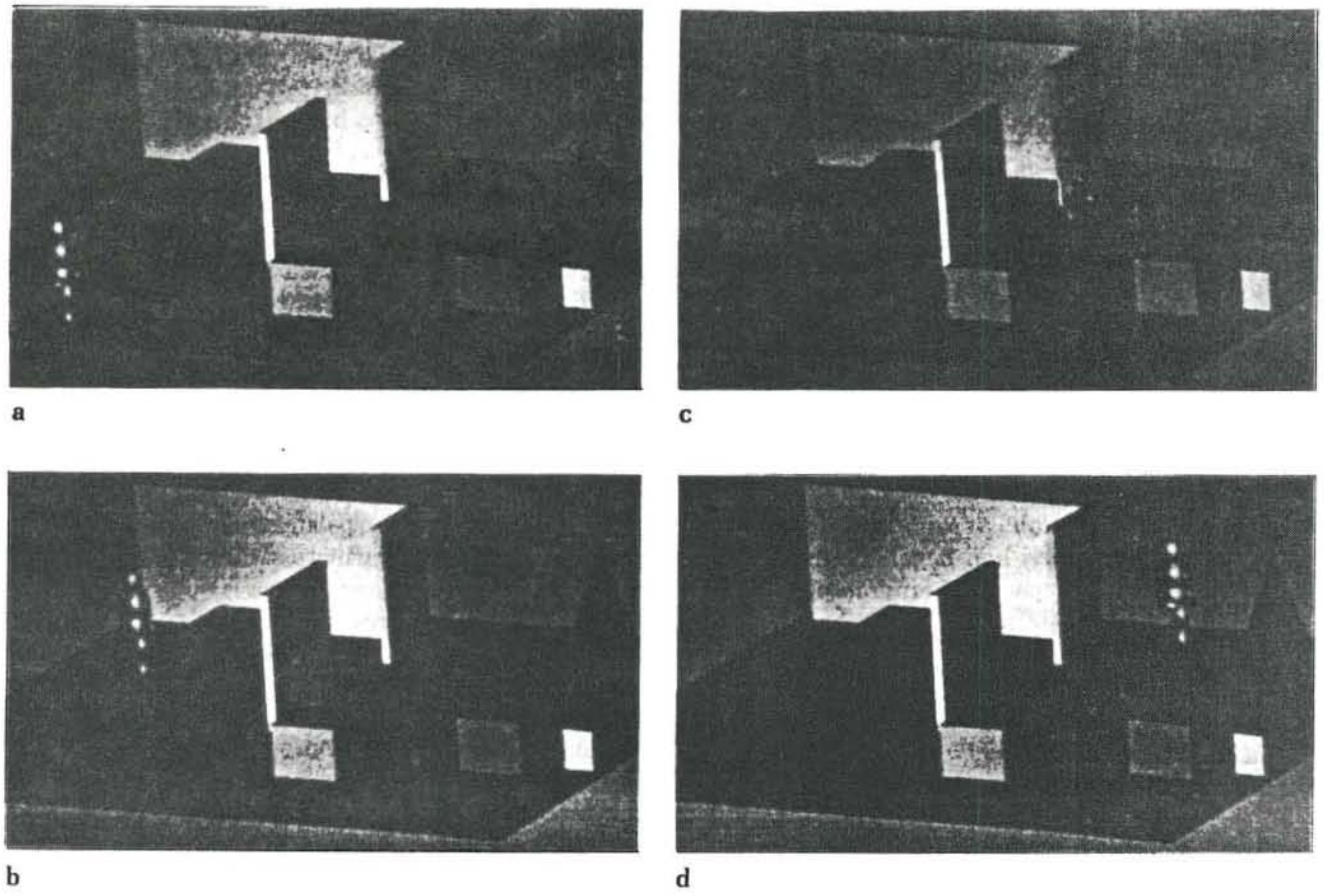


Figure 5. Path planning in a static environment.

positions can be used to define the motion between the trajectory points.[20,26]

An important part of trajectory planning is collision detection and obstacle avoidance.[41] A simple algorithm for obstacle avoidance assumes a straight line path from start to goal and tests for possible collisions. When a collision is detected, a new path that avoids the collision can be generated. This process is repeated until a collision-free path is found. Collision detection can be done by calculating a sweep volume, which tests for overlap between the moving object and the obstacles, or by testing for collisions with a point on the object and scaling up the obstacles to account for object size.

Collision detection performed iteratively by intersection testing may not be particularly helpful in choosing an optimum path, since only local information is used to generate each new path. An alternative is to specify all constraints on the vertices of the moving object solving for a path that obeys constraints simultaneously.

The Lozano-Perez algorithm[41] creates a graph whose nodes are the starting and goal positions and boundary points on the obstacles. The edges between these nodes are weighted according to the straight-line distance between these positions. If the line joining positions passes through an obstacle, its weight is some maximal value. A minimal path around obstacles can be found using the Dijkstra algorithm. Figure 5 illustrates this algorithm which is used to find a path for a body through a complex environment. Four subgoals are used to make the path more interesting.

Path planning is made more complex if positional constraints do not all apply simultaneously or when obstacles are themselves in motion. In this case, collision avoidance must be done on the fly. This problem is discussed in the section covering stimulus-response control.

Algorithmic control

Although all computer-generated motion is controlled by algorithms, "algorithmic motion control" refers to generation of motion using a series of preprogrammed instructions with minimal user input concerning actual motion. The user may suggest parameters controlling the algorithm, but does not control the specific motion at individual degrees of freedom. Simple algorithmic control, often seen in computer-generated animation, includes spinning cubes, planets following elliptical revolutions, and other cases of repetitive motion.

Algorithms used to generate still images can often be used to develop interesting animations by recording the generation process. The organic forms generated by Kawaguchi[34] are examples. The regularity and order of many natural forms, such as shells, horns, and branching plants, can be represented by mathematical expressions. For example, shells are formed by twisting and expanding a basic polyhedral model. Plants can be formed in a similar manner, with additional consideration for the amount and angle of branching and the diameter of branches. By recording the generation in progress, these bodies can be seen to grow.

Stochastic control

An algorithmic technique that is becoming common in computer modeling is use of stochastic processes to generate objects such as mountains,[33] trees,[42] clouds,[43,44] glass, waves, and bagels.[45] Stochastic has come to mean use of (pseudo-) random perturbations applied during modeling to create more interesting and/or realistic images. Though these techniques are often used for still images, they can also be used to generate interesting motion. A prime example is fractal, or stochastic, mountain generation.[33] Fractal landforms are created by subdividing constituent polygons with perturbation of vertices. If the process of subdivision is displayed over time, a fairly realistic animation of mountain building is produced, as seen in the Genesis Demo in the film *Star Trek II: The Wrath of Khan*, (Paramount 1982).

A related example is Reeves' *particle systems*.[27,28] Large numbers of particles can be used to model many natural phenomena, such as clouds, fire, atmospheric disturbances, and vegetation. Particles are easy to render, but they incorporate in simple form many sophisticated features. They are controlled by a number of attributes, such as position, velocity, size, color, and shape. Particles can be given a limited lifetime so that some die (disappear) and others are born (appear) during the animation.

Stochastic processes are used to create and alter each particle's shape, motion, and appearance. The user can specify parameters such as mean number of particles generated per frame, where they are placed, and how they initially move, but the actual motion is not determinate. Particles can be used to represent light sources, as in the imitation of fire in the movie *Star Trek II*. By drawing the path of each particle over its entire lifetime, vegetation can be modeled, as in Alvy Ray Smith's *white sand*.[27]

A more sophisticated version of structured particle systems[28] convincingly models more complex objects such as a forest, and more complex motion, such as grass blowing in the wind. The use of stochastically generated "wind particles" to simulate grasses blowing is particularly interesting from a motion control point of view. It suggests that stochastic motion might hold potential for modeling a wide range of natural motions, such as sea creatures and plants buffeted by waves, airborne seeds, protozoa, and insects.

Stimulus-response behavioral control

Stimulus-response control suggests that environmental interactions are taken into account during motion generation. The motion of each object is dependent not only on its own internal algorithm, but on the behavior of other objects. Stimulus-response control could be added onto a stochastic algorithmic control algorithm. For example, it could be added to the algorithm for tree growth by also taking into account the changing environment of the forest: the sun, proximity of neighboring plants, people with axes, etc. Biologically speaking, the fundamental growth algorithm is genetically programmed, but the exact pattern of the growth is environmentally determined.

Stimulus-response control, obviously, involves two steps: recognizing the state of the environment and developing a response to it. To explore this, a simple system using spheres has been developed at the University of California, Santa Cruz, Computer Graphics and Imaging Laboratories. Spheres can be given a start position, end position, and velocity, as well as a color and size. When let loose in their environment, they move toward the destination, checking for collisions with other spheres.

Figure 6 illustrates initially green and white spheres on the move. When they collide and pass through each other, they turn red; when they reach their destination, they turn blue. Various collision avoidance algorithms can be explored using this simple system, as can more sophisticated types of interaction. Collision avoidance is not shown, as it is not particularly illustrative in still images.

Another example of stimulus-response control was a project unofficially called "fishbrains" at the Atari Research Labs under Ann Marion.[4] A small environment with a variety of creatures was modeled. Characters' responses to each other depended both upon their own internal states, such as hunger, fatigue, or calm, and the creatures encountered.

Learning

At an even more sophisticated level, it is possible to imagine motion control algorithms that learn. Learning is a difficult problem that is not well understood, either for machines or animals. Machine-learning algorithms that have been developed in robotics and artificial intelligence[20,46] provide insight into the use of learning for motion control. On the other hand, computer animation may provide an excellent test site for the practicality of these learning algorithms.

Fairly simple learning algorithms based on common sense may provide a starting point. A simple gauge that registers success or failure by whether the desired goal is reached or how many collisions occur, could be used to alter future motion responses. For example, a simulated fish attempting to avoid predators whose motion is repetitive, might have to learn to respond correctly to avoid being eaten. If the predator typically attacks from underneath the prey, the prey should attempt to get beneath the predator.

Conclusions

Over the past several years, the trend toward greater complexity and more realistic static scenes has been greatly aided by knowledge from other fields such as physics. A similar process is now occurring in the field of motion control for computer animation. Motion is becoming progressively more complex and realistic. Control of motion can be greatly aided by integrating knowledge from related fields such as robotics, artificial intelligence, psychology, biology, and physics. Use of more automatic motion control techniques, such as automatic path planning, collision detection, dynamic analysis, stochastic algorithms, stimulus-driven response, and learning algorithms offer great potential for animating highly complex motion with minimal user input.■

Acknowledgments

I wish to thank Michael Berman, Jane Johnson, Cathryn Nelson, Ken Cluff, and Al Conrad for their assistance in developing and implementing the algorithms illustrated in this article.

References

1. P. Hanrahan and D. Sturman, "Interactive Animation of Parametric Models," *Introduction to Computer Animation*, course notes, ACM SIGGRAPH 85, Aug. 1985, pp. 87-101.
2. D. Zeltzer, "Motor Control Techniques for Figure Animation," *IEEE CG&A*, Nov. 1982, pp. 53-60.
3. F. Parke, "Parameterized Models for Facial Animation," *IEEE CG&A*, Nov. 1982, pp. 61-68.
4. C.W. Reynolds, "Description and Control of Time and Dynamics in Computer Animation," *Advanced Computer Animation*, course notes, ACM SIGGRAPH 85, July 1985, pp. 289-296.
5. E. Catmull, "The Problems of Computer-Assisted Animation," *Computer Graphics* (Proc. SIGGRAPH 78), July 1978, pp. 348-353.

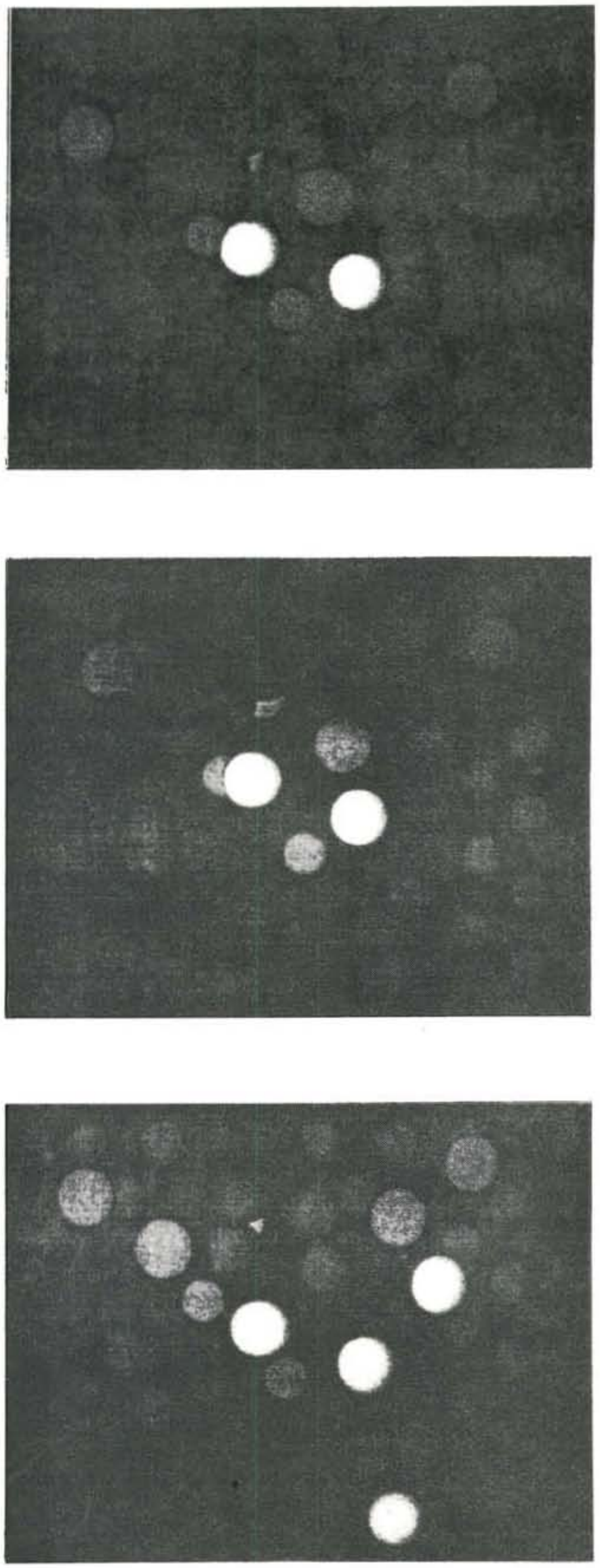


Figure 6. Collision detection in a changing environment.

6. W.T. Reeves, "Inbetweening for Computer Animation Using Moving Point Constraints," *Computer Graphics* (Proc. SIGGRAPH 81), Aug. 1981, pp. 263-269.

7. S.N. Skeketee and N.I. Badler, "Parametric Keyframe Interpolation Incorporating Kinetic Adjustment and Phrasing Control," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 255-262.

8. R.M. Baecker, "Picture-Driven Animation," *Proc. Spring Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 1969, pp. 273-288.

9. K. Shelley and D. Greenberg, "Path Specification and Path Coherence," *Computer Graphics* (Proc. SIGGRAPH 82), July 1982, pp. 157-166.

10. J. Wilhelms, "Virya-A Motion Control Editor for Kinematic and Dynamic Animation," *Proc. Graphics Interface 86*, Morgan Kaufmann, Inc., Los Altos, Calif., May 1986, pp. 141-146.

11. C.W. Reynolds, "Computer Animation with Scripts and Actors," *Computer Graphics* (Proc. SIGGRAPH 82), July 1982, pp. 289-296.

12. N. Magnenat-Thalmann and D. Thalmann, "The Use of 3D High-Level Graphical Types in the MIRA Animation System," *IEEE CG&A*, Dec. 1983, pp. 9-16.

13. T.J. O'Donnell and A.J. Olson, "GRAMPS—A Graphics Language Interpreter for Real-time, Interactive, Three-Dimensional Picture Editing and Animation," *Computer Graphics* (Proc. SIGGRAPH 81), Aug. 1981, pp. 133-142.

14. A. Hutchinson Guest, *Dance Notation*, Dance Books, London, 1984.

15. G. Politis and D. Herbison-Evans, "A Computer Graphics Interpreter for Benesh Movement Notation," *Proc. Ausgraph 85*, 1985, pp. 25-30.

16. B. Singh et al., "A Graphical Editor for Benesh Movement Notation," *Computer Graphics* (Proc. SIGGRAPH 83), July 1983, pp. 51-62.

17. N.I. Badler, J. O'Rourke, and B. Kaufman, "Special Problems in Human Movement Simulation," *Computer Graphics* (Proc. SIGGRAPH 80), July 1980, pp. 189-203.

18. T.W. Calvert, J. Chapman, and A. Patla, "The Integration of Subjective and Objective Data in the Animation of Human Movement," *Computer Graphics* (Proc. SIGGRAPH 80), July 1980, pp. 198-203.

19. G.J. Savage and J.M. Officer, "CHOREO: An Interactive Computer Model for Dance," *Int'l J. Man-Machine Studies*, July 1978, pp. 1-3.

20. C.S. George Lee, R.C. Gonzalez, and K.S. Fu, *Tutorial on Robotics*, IEEE Computer Soc. Press, Silver Spring, Md., 1983.

21. R.P. Paul, "WAVE: A Model-Based Language for Manipulator Control," Tech. Report MR76-615, Soc. of Manufacturing Engineers, Dearborn, Mich., 1976.

22. R. Finkel et al., "An Overview of AL, a Programming Language for Automation," *Proc. Fourth Int'l Joint Conf. Artificial Intelligence*, pp. 758-765.

23. L. Lieberman and M. Wesley, "AUTOPASS: An Automatic Programming System for Computer-Controlled Mechanical Assembly," *IBM J. Research and Development*, Vol. 21, No. 4, 1977, pp. 321-333.

24. R. Tomovic, "On Man-Machine Control," *Automatica*, Pergamon Press, Oxford, 1967.

25. J. Wilhelms, "Graphical Simulation of the Motion of Articulated Bodies such as Humans and Robots, with Particular Emphasis on the Use of Dynamic Analysis," doctoral dissertation, Computer Science Div., Univ. of California, Berkeley, Calif., July 1985.

26. R.P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*, The MIT Press, Cambridge, Mass., 1981.

27. W.T. Reeves, "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects," *Computer Graphics* (Proc. SIGGRAPH 83), July 1983, pp. 359-376.

28. W.T. Reeves and R. Blau, "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particles Systems," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 313-322.

29. J. Weil, "The Synthesis of Cloth Objects," *Computer Graphics* (Proc. SIGGRAPH 86), Aug. 1986, pp. 49-54.

30. T.W. Calvert, J. Chapman, and A. Patla, "Aspects of the Kinematic Simulation of Human Movement," *IEEE CG&A*, Nov. 1982, pp. 41-52.

31. D. Herbison-Evans, "Nudes 2: A Numeric Utility Displaying Ellipsoid Solids," *Computer Graphics* (Proc. SIGGRAPH 78), Aug. 1978, pp. 354-356.

32. J. Wilhelms and B.A. Barsky, "Using Dynamic Analysis for the Animation of Articulated Bodies such as Humans and Robots," *Proc. Graphics Interface 85*,, Canadian Information Processing Soc., Toronto, Ont., May 1985, pp. 97-104.

33. A. Fournier, D. Fussel, and L. Carpenter, "Computer Rendering of Stochastic Models," *Comm. ACM*, June 1982, pp. 371-384.

34. Y. Kawaguchi, "A Morphological Study of the Form of Nature," *Computer Graphics* (Proc. SIGGRAPH 82), July 1982, pp. 223-232.

35. A.R. Smith, "Plants, Fractals, and Formal Languages," *Computer Graphics* (Proc. SIGGRAPH 84), July 1984, pp. 1-10.

36. J.U. Korein and N.I. Badler, "Techniques for Generating the Goal-Directed Motion of Articulated Structures," *IEEE CG&A*, Nov. 1982, pp. 71-81.

37. K.D. Willmert, "Visualizing Human Body Motion Simulations," *IEEE CG&A*, Nov. 1982, pp. 35-43.

38. R.V. Lundin, "Motion Simulation," *Proc. Nicograph 1984*, Nov. 1984, pp. 2-10.

39. M. Girard and A.A. Maciejewski, "Computational Modeling for the Computer Animation of Legged Figures," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 263-270.

40. W.W. Armstrong and M.W. Green, "The Dynamics of Articulated Rigid Bodies for Purposes of Animation," *Proc. Graphics Interface 85*, Canadian Information Processing Soc., Toronto, Ont., May 1985, pp. 407-415.

41. T. Lozano-Perez and M.A. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *Comm. ACM*, Oct. 1979, pp. 560-570.

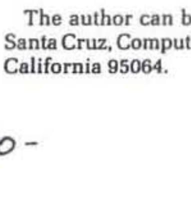
42. J. Bloomenthal, "Modeling the Mighty Maple," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 305-311.

43. R. Voss, "Fourier Synthesis of Gaussian Fractals: L/F Noises, Landscapes and Flakes," *State of the Art in Image Synthesis*, course notes, ACM SIGGRAPH 83, July 1983.

44. G.Y. Gardner, "Visual Simulation of Clouds," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 297-303.

45. K. Perlin, "An Image Synthesizer," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 287-296.

46. S. Tangwongsan and K.S. Fu, "An Application of Learning to Robotic Planning," *Int'l J. Computer and Information Science*, Vol. 8, 1979, pp. 303-333.



Jane Wilhelms has been an assistant professor with the Computer and Information Sciences Board at the University of California, Santa Cruz, since 1985. Her research interests include computer animation, modeling articulated bodies, and use of dynamic analysis for motion control.

Wilhelms received a BA in zoology from the University of Wisconsin, Madison, an MA in biology from Stanford University, and an MS and a PhD in computer science from the University of California, Berkeley. She is a member of IEEE and ACM.

The author can be contacted at the University of California, Santa Cruz, Computer & Information Sciences Board, Santa Cruz, California 95064.

Using Dynamic Analysis for Realistic Animation of Articulated Bodies

Jane Wilhelms
Computer Graphics and Imaging Laboratory
Computer and Information Sciences
University of California, Santa Cruz, CA 95018

ABSTRACT

A major problem in computer animation is creating motion that appears natural and realistic, particularly in the case of complex articulated bodies such as humans and other animals. At present, truly lifelike motion is produced mainly by copying recorded images, a tedious and lengthy process requiring considerable external equipment. An alternative that is explored here is the use of *dynamic analysis* to predict realistic motion. Using dynamic motion control, bodies are treated as masses acting under the influence of external and internal forces and torques. Dynamic control is advantageous because motion is more naturally restricted to physically-realizable patterns, and many types of motion can be automatically predicted. Use of dynamics suffers from problems of computational cost and the difficulty of specifying controlling forces and torques. However, evidence has accumulated that dynamics does offer hope for more realistic, natural, and automatic motion control. Because such motion simulates real world conditions, an animation system using dynamic analysis is also a useful tool in such related fields as robotics and biomechanics.

1. Introduction

The major problem facing computer animation is the creation of motion that appears natural and realistic, particularly when the objects in motion are complex articulated bodies such as animals or when multiple objects interact. The problem involves a number of interesting aspects: e.g., what kind of user-interface allows motion specification with the least effort, how can the animation system incorporate the intelligence to interpret general motion descriptions and translate them into specific motion instructions, how can motion be restricted to desirable patterns, and how can motion be best described at the lowest level of positions taken over time?

This work was supported at the University of California, Santa Cruz, by National Science Foundation grant number CCR-8606519 and UCSC fellowship 660177-19900. The work was supported at the University of California, Berkeley, by National Science Foundation grant numbers ECS-8204381 and DCI-8451997 and the Defense Advanced Research Projects Agency contract numbers N00039-82-C-0235 and N00039-84-C-0089.

Most animation systems are *kinematically-based*, in the sense that the motion description, from user-input through actual frame rendering, consists of positions specified over time.[1,2,3,4,5] On the other hand, motion in the real world is *dynamically-based*, determined by a complex interplay of forces and torques acting on masses. This paper explores the use of *dynamic analysis* to generate animation of articulated bodies. Dynamic analysis involves setting up and solving the dynamics equations of motion relating the forces and torques acting on masses to their acceleration.[6] Given forces and torques acting on a body, dynamic analysis provides the accelerations that the parts of the body will undergo. Using integration, these accelerations can be used to find new positions, thus generating motion.

Section 2 compares kinematic and dynamic animation systems. Section 3 provides some background in dynamic analysis. Section 4 discuss controlling motion using dynamics. Section 5 discusses a graphical editor for specifying motion when using dynamics and walks the reader through a sample session of generating a dynamic animation. Section 6 discusses problems with using dynamic motion control and possible solutions. Throughout the paper, a dynamically-based animation system, *Deva*, is used to illustrate the method.

2. Kinematics Versus Dynamics for Animation

Kinematic systems force the user to choose the desired motion from a tremendous variety of possible motions. The most successful animations of living creatures, in terms of the reality of output, rely on copying motion from recorded images, either by *rotoscoping* (actually tracing the image) or measuring the positions taken in each recorded frame and using them to drive the objects being animated. This approach has a number of difficulties: the need for external recording and tracing equipment, the fact that motion is highly specific and cannot be altered with impunity, the time it takes to measure the recorded frames, and the impossibility of finding motion descriptions for imaginary animals or conditions where recordings aren't available.

A simpler, though generally less successful kinematic approach is reliance on user input facilitated by an interactive system. Often this method uses *3-D keyframing* where the user positions the body as desired at specified times, and motion is interpolated between these positions for animation.[7,8] While 3-D keyframing is more successful than 2-D keyframing (not suffering from the inherent information loss), it still relies upon the user's ability to designate the appropriate path and rate of motion. Constraint-based systems aid in limiting the range and type of motion, but do not provide a complete solution to the problem of making motion appear realistic.[9]

An alternative is to give the world being animated greater physical reality, including descriptions of masses and active forces and torques, and use dynamic analysis to predict motion. Use of dynamics in animating articulated bodies for computer graphics had largely been limited to crash studies,[10] though recently work has begun to appear utilizing dynamics for bodies under internal control.[11,12,13,14] Dynamics is also used in CAD/CAM and robotic applications, such as simulating and controlling vehicles or mechanical manipulators[15,16,17,18] and software packages capable of doing dynamic analysis such as DRAM are available.

Dynamically-predicted motion is advantageous because motion can be naturally restricted to realizable patterns and many types of motion, such as falling or reacting to collisions, can be found automatically. It is possible for the user to specify motion at a limited subset of body joints and have the dynamics calculate appropriate motion at the rest. The problems of inverse kinematics (finding local joint positions that place the body in the desired world space configuration) can be dealt with by pushing and pulling with external forces. Positional constraints can be simulated using appropriate forces holding the body in position.[19]

Dynamic analysis suffers from three main problems: first, the cost of the analysis; second, numerical instability when the bodies are complex; and third, controlling the motion. The control problems are somewhat similar to the motion specification problem found in strictly kinematic systems, but even more related to control problems in robotics.[16, 20, 18] Simulating articulated body motion with dynamics is a difficult problem because of the many degrees of freedom, the complex coordinated motion possible, and our unwillingness to accept mildly unrealistic motion for bodies are familiar as humans and animals. Much of what is discussed here is equally relevant to the problem of dynamically-animating simpler bodies.

The ability to realistically simulate worlds using computer animation is also important in other fields. Many issues in motion control for computer graphics have parallels in robotics,[16, 20] and simulation systems are important in developing tools for designing and testing robots and other manipulators. In biomechanics and medicine, graphical simulations provide a means of analyzing motion and testing mathematical models of animal motion.[21, 22, 23] In sports, simulations can aid in developing more efficient and safe ways of moving.[24]

3. Background in Dynamics

Dynamic analysis refers to the study of the relationship between forces (for translational motion) and torques (for revolute motion) and the motion of masses, a relation which is expressed by the *dynamics equations of motion*.[6] While a number of different dynamics formulations are available, they can all be seen as variations on the simplest, most familiar form known as *Newton's Second Law*,[25]

$$\mathbf{F} = m\mathbf{a}$$

or, the force $\mathbf{F}$ acting on a particle is equal to its mass m times its acceleration $\mathbf{a}$. The dynamics equations increase in complexity when the masses involved are extended bodies not points and when multiple masses interact, but the fundamental concept of relating forces and torques to accelerations remains.

The number of dynamics equations necessary to specify a system depends on the number of *degrees of freedom* of the system. A point mass has three translational degrees of freedom in space. An extended rigid body free to move in space has 6 degrees of freedom, three translational and three revolute. In the case of articulated bodies, those in which masses are connected by joints, the number of degrees of freedom is the sum of the degrees of freedom at each joint plus the number of degrees of freedom connecting the body to the world. Thus, a human body simplified to 18 internal degrees of freedom and free to move in the world would be described by 24 dynamics equations. Because of the interaction taking place between masses in an articulated body, the equations are complex and coupled.[16] This means that the intuitive idea that one can control the body dynamically by merely considering the torques acting between neighboring segments independently is incorrect. A torque acting on the shoulder will affect to varying extents the elbow, wrist, trunk, and the rest of the body.

Though all dynamics formulations express the behavior of the world being modeled and give the same result, their appropriateness depends upon the problem being analyzed. The Newtonian formulation is common, but other formulations such as the Lagrangian, the Gibbs-Appell,[26] and some recursive methods[27, 15, 17] are sometimes more appropriate for describing articulated bodies.

Although this paper describes a system, *Deva*, which uses the Gibbs-Appell formulation, the author has become convinced of its inferiority compared to the much faster recursive methods such as that of Armstrong.[27] The Gibbs-Appell formulation has a cost of $O(n^4)$ for n degrees of freedom, while the Armstrong method is linear in n ($O(n)$). At present work is proceeding on the interactive system, *Manikin*, [19] which uses the Armstrong formulation to explore use of dynamics for positioning and manipulating articulated bodies. The Gibbs-Appell-based system *Deva* is described in this paper because the control and environmental simulation it uses are more sophisticated than those available in *Manikin* and because the Gibbs-Appell formulation neatly and elegantly partitions the complex force, torque, mass, configuration, and velocity information needed to do dynamics in a way somewhat more clear than the recursive method. It is hoped that readers can map the control methods described here onto faster dynamics formulations without great trouble. The possibility also exists that a recursive Gibbs-Appell formulation can be developed, as it has been shown[28] that it is possible to find recursive versions of non-recursive formulations.

Another difference between the formulations is that the Armstrong method assumes that all joints have three revolute degrees of freedom and that the body is connected to the world by six degrees of freedom, thus, non-spherical joints must be otherwise constrained and sliding joints cannot be modeled. The Gibbs-Appell formulation allows for more general joints. Featherstone[15] has also described a recursive linear dynamics formulation without this restriction on the type of joints.

3.1. The Gibbs-Appell Dynamics Formulation

This section provides an overview of the Gibbs-Appell formulation. Its derivation and details of its use are given in Appendix I and elsewhere[29, 13, 14]

For simplicity, a body is represented by segments consisting of rigid extended masses connected by joints of from one to six degrees of freedom. The body segments are described as a tree structure branching from the fixed, massless world segment. The initial joint connects the body to the world. For a body free to move in space, this initial joint has six degrees of freedom. For purposes of dynamics, multiple-degree-of-freedom joints, such as the shoulder, are treated as a sequence of one-degree-of-freedom joints joined by massless, dimensionless segments.[30]

Modeling such joints as a sequence means that rotations are *euler*. Taking as an example the spherical shoulder joint with three rotary degrees of freedom and assuming the sequence is an x-rotation followed by a y-rotation followed by a z-rotation, the y-

rotation is not about the fixed-joint-coordinate-frame y-axis, but around a new y-axis created by rotating that frame about the x-axis. The negative consequence of this is that the meaning of the y-rotation is not intuitively obvious; the positive consequence is that if the initial z-axis is along the longitudinal segment axis, it remains so. The alternative model of the spherical joint, in which all three rotations are about the fixed joint coordinate frame, requires that longitudinal rotations be mapped to rotation about an arbitrary axis. In terms of user convenience, it is perhaps best to allow the user to specify the three rotations as an x- and y-rotation relative to the fixed joint frame and a final rotation about the longitudinal segment axis. It is not difficult to map this to either the Euler or the fixed coordinate model.[31]

The dynamics equations express the *generalized force* at each degree of freedom as a function of the mass distribution, acceleration, and velocity of all segments distal to this degree of freedom. The generalized force can be thought of as the net force or torque (depending on whether the degree of freedom is sliding or revolute) active at this degree of freedom and is the result of a combination of gravitational, frictional, constraint, controlling, and applied forces and torques active within the system.

For a sliding joint r, the generalized force is

$$q_r = \sum_{k=r}^{distal} (m_k \mathbf{a}_k^T (\frac{\partial \mathbf{a}_k}{\partial \ddot{s}_r}))$$

For a revolute joint r, the generalized force is

$$q_r = \sum_{k=r}^{distal} (m_k \mathbf{a}_k^T (\frac{\partial \mathbf{a}_k}{\partial \ddot{\theta}_r}) + \alpha_k^T \mathbf{I}_k (\frac{\partial \alpha_k}{\partial \ddot{\theta}_r}) + (\frac{\partial \alpha_k}{\partial \ddot{\theta}_r})^T (\omega_k \times \mathbf{I}_k \omega_k))$$

where, for segment k,

q_k = generalized force

m_k = segment mass

$\mathbf{a}_k$ = worldspace acceleration vector of the center of mass

$\mathbf{a}_k^T$ = transpose of acceleration vector of the center of mass

α_k = worldspace angular acceleration vector

ω_k = worldspace angular velocity vector

$\mathbf{I}_k$ = inertial tensor describing mass distribution

$\ddot{\theta}$ = local angular acceleration

$\ddot{s}$ = local acceleration

Because each segment's worldspace acceleration (and, similarly, angular acceleration, velocity, and configuration) is just a function of its local acceleration relative to its parent segment and the worldspace acceleration of the parent relative to the world, these equations can be expressed in terms of purely local motion and configuration. Rearranged into a matrix formulation, they can be simply stated as

$$\mathbf{M}\,\ddot{\mathbf{c}} + \mathbf{V} = \mathbf{q} \qquad \text{or} \qquad \mathbf{M}^{-1}(\mathbf{q} - \mathbf{V}) = \ddot{\mathbf{c}}$$

For a body with n degrees of freedom, $\ddot{\mathbf{c}}$ is an n-length vector of the local angular or linear acceleration at each degree of freedom (local referring to motion of the segment distal to this degree of freedom relative to the proximal segment) and $\mathbf{q}$ is an n-length vector specifying the generalized force active at each degree of freedom. $\mathbf{M}$ is an $n \times n$ inertial matrix describing the configuration of the system masses. $\mathbf{V}$ is an n-length vector dependent upon the configuration of the masses and their velocity relative to each other. Thus, if the generalized force vector $\mathbf{q}$ is known, the equations can be solved for accelerations $\ddot{\mathbf{c}}$. This is known as *indirect dynamics* and is the direction used to find motion for animation. Alternatively, if the accelerations $\ddot{\mathbf{c}}$ are known, the equations can be solved for the generalized forces $\mathbf{q}$. This is known as *direct dynamics* and is the direction commonly used in robotics.[16] The equations used to find these vectors and the matrix are found in Appendix I.

In solving the equations for accelerations, the mass is known and assumed to be unchanging for a given body, the mass distribution is automatically calculated based upon present configuration, and the present known velocities are used. Given generalized forces, this leaves a set of linear equations which can be solved using Gaussian elimination for local accelerations. Numerical integration techniques, such as the Runge-Kutte,[32] can then be used to find new velocities and positions.

4. Causes and Control of Motion Using Dynamics

The forces and torques responsible for motion come from a variety of external and internal sources (see Figure 1):

(1) Some forces and torques can be accurately and automatically calculated, such as those due to gravity or to known external applied pushes or pulls. Realistic animation of this type of motion can be generated with minimal user input.

(2) Some forces and torques can be simulated within the system, [33] such as those due to joint limits, the ground and collisions with other objects. Springs and dampers can be used to model these restrictions, but finding appropriate springs and dampers to simulate the motion of segments with widely varying masses and motion is not simple and they tend to create numerical instability in the dynamics equations. However, acceptable methods have been developed to model these restrictions with little user input.

(3) The crux of the dynamic control problem lies in finding the non-automatic forces and torques. These include those *controlling forces* and *torques* which simulate muscles in animals or motors in robots. *Deva*'s ability to generate arbitrary controlled motion using dynamics is still low-level and primitive.

4.1. Automatically Calculated Forces and Torques

Automatic forces and torques presently implemented in *Deva* include those due to gravity, joint limits, and the ground. Detailed descriptions of how these are calculated can be found in Appendix II and elsewhere.[34, 14]

Gravity is calculated using the known mass distribution of body segments. For each translational degree of freedom, the effect of gravity is found by summing the gravitational forces acting through the centers of mass of all segments distal to this degree of freedom and finding the component of this summed force acting along the axis of sliding of this degree of freedom. For revolute joints, gravitational torque must be considered, torque being the product of a force times the perpendicular distance between the point of application of the force and the axis of rotation. The gravitational torque at a revolute degree of freedom is found by summing the torque due to gravity at each distal segment and finding the component of this torque acting along the axis of rotation. By altering the gravitational acceleration, motion on other planets or in space can be easily simulated. Figure 2 shows a body falling freely through space under the influence of gravity.

Joint Limits. While constraints included in the dynamics equations automatically restrict motion to the allowed degrees of freedom (e.g., if the knee is described as a one degree-of-freedom hinge joint it cannot rotate sideways or longitudinally), dynamics does not automatically restrict motion within a particular degree of freedom. Thus, if an arm is dropped the elbow can bend backwards unnaturally or the lower arm may fall through the upper (see Figure 3). Kinematically this problem is solved by using joint limits beyond which motion cannot occur. Dynamically, a rigid limit is undesirable because it does not mimic the natural springy motion that sometimes occurs when such positions are reached.

For *Deva*, joint limits are simulated using a combination of a spring and damper at each degree of freedom. A spring supplies a counteracting force or torque proportional to the amount past the specified limit the joint has deformed. The damper supplies a counteracting force or torque proportional to the velocity at this degree of freedom. The strength of a spring and damper could be proportioned simply by using a spring and damper constant which takes into account particular conditions at the joint. To avoid having to determine appropriate constants for each degree of freedom, these constants are replaced in *Deva* with a variable dependent upon the mass distribution and/or velocity of segments distal to the degree of freedom. While this value normally restricts motion appropriately, it can also be controlled by a proportionality factor input by the user. Figure 4 shows an arm lifted by a torque at the shoulder and elbow and dropped. Note that as it falls the elbow reaches an angle of 180 degrees relative to the upper arm and is restricted there by the joint limit.

Ground Reaction Forces. Dynamics does not automatically take into account external environmental constraints such as remaining on or above the ground (see Figure 1). This problem is treated in *Deva* by modeling the ground reaction forces as external applied forces due to springs and dampers. To detect collisions with the ground, the locations of the vertices of a max-min box surrounding each segment are compared to a horizontal flat ground plane (a more complex ground would be possible with more sophisticated collision detection). Since dynamics is a numerical process, it is possible for these vertices to have descended below the ground during the previous time sample. Normally this is not a noticeable problem; when it is, supersampling can minimize it.

The ground reaction force actually has three perpendicular components: a vertical reaction force counteracting motion into the ground, and two horizontal reaction forces mimicking friction along the ground. The vertical component consists of a reaction spring that is a function of the amount below the floor each point is descended and a reaction damper which is a function of the velocity during penetration. These tiny springs and dampers are applied to oppose each vertex that has penetrated the ground. Though the reaction force is automatically calculated, it can again be scaled by user input to simulate a ground which is hard or soft, rigid or elastic. A weak spring can also simulate the buoyancy of water. The horizontal reaction force due to friction is taken to be a fraction of the vertical reaction force if there is horizontal motion. Otherwise there is no horizontal force. By varying the frictional force, a ground that is slippery like ice or rough and sticky can be simulated. (See Figures 5 and 6.)

Other Automatic Forces and Torques. While not implemented in *Deva*, other forces and torques can be simulated by techniques similar to those described above. At present, no internal collision detection is done, so that the body can move through itself (e.g., if the body is falling forward, the arms could swing through the legs). This could be handled by collision detection and simulated with springs and dampers in the same manner as floors. A combination of collision detection and applied springs and dampers can also simulate environmental collisions with walls, tables, etc. This is with the caveat that while the collision may be realistic for the model, it would not take into account such things as reflex reactions to collisions on the part of the animal, stiffening of muscles, etc. While theoretically this could be included in the model, the control problems would be considerable. An external applied force, such as being pulled by a rope, can be modeled similar to the gravitational force, except that its point of application, magnitude, and direction are under user control.

The forces and torques described above allow the automatic prediction of object behavior excluding internal muscular or motor control. Thus, the complex motions of falling, colliding with the ground, etc. can easily be simulated using dynamic analysis.

4.2. Controlling Forces and Torques

Given that many environmental influences are automatically taken into account using dynamic motion control, the difficulty then becomes finding and coordinating the internal forces or torques (simulating muscles in animals or motors in machines) to cause a specific desired motion. The controlling force (for translational degrees of freedom) or torque (for revolute degrees of freedom) is treated in *Deva* as an ideal actuator acting only on one particular degree of freedom. This avoids the problem of dealing with actual muscles whose torques vary with angle of rotation, and which can affect more than one degree of freedom and even cross more than one body segment. For biomechanical studies, it would be possible to add a preprocessor that takes muscular forces and torques and converts them to this simple actuator form.

The determination of controlling forces and torques in *Deva* is still low-level, in that while the user has a fair amount of control over the state of individual degrees of freedom, he or she has little ability to specify coordination between joints. At present, each degree of freedom can exist in one of five different states: 1. *direct dynamic control*: a specific controlling force or torque is designated; 2. *relaxation*: no controlling force or torque is applied; 3. *frozen*: local position is maintained; 4. *oriented*: world-space orientation is maintained; or 5. *hybrid K-D control*: desired positions over time are specified by the user and the system attempts to find forces and torques to achieve these positions. A degree of freedom can only be in one of these states at a time, but can alternate between states during the animation.

1. Direct dynamic control: Controlling forces and torques can be provided by the user (using the graphical editor *Virya* described below) as functions of force or torque versus time for each degree of freedom. These functions are modeled as cubic interpolatory splines to ensure second derivative continuity and sampled to find the contribution of controlling forces and torques for each degree of freedom.[35, 36] The arm shown in Figure 4 is lifted using this method by applying a torque at the shoulder. While dynamically acceptable, this method involves (and even accentuates) the problems found in kinematic motion control; i.e., the user must specify a force or torque for each degree of freedom being internally controlled, as well as predict what force or torque will provide the desired motion. This latter problem is even more difficult and unnatural than specifying motion kinematically, though it is lessened by the fact that the user need only specify a controlling force or torque for those joints under internal control. robotics.

Direct dynamic control can be useful in modeling world-space interactions, such as pulling the body toward a goal or pushing on it with an external force.

2. *Relaxation*: Relaxation is a default state in which the degree of freedom hangs loose and reacts to external conditions, joint limits, and the motion of the rest of the body. The arm in Figure 4 falls after being lifted because the joints are reset to the relaxed state.

3. *Freezing*: In many controlled motions, some joints are held locally fixed; e.g., in Figure 4 the elbow is frozen while the shoulder is raised. Because of the dynamic interaction between body segments, controlling forces and torques must be applied to provide this local stability. Normally, the animal body uses feedback to accomplish this, altering forces and torques as necessary to maintain position. The freeze function in *Deva* uses the simple method of clamping the joint end limits around the present position, so that a spring and damper are applied when the joint moves from its present position. This method has the advantage over rigidly holding the joint in position that it does not interfere with the overall dynamic interaction and some small amount of natural motion is still possible about the frozen joint. (Freezing could also be done using hybrid control and specifying a static position.)

4. *World-Space Orientation*: A related feature is the orienting function. While freezing stabilizes a segment locally at its proximal joint, orientation stabilizes a segment in a designated world space orientation. This is a first attempt at dealing with the problems of balance. Orientation is provided by the user in the form of an orientation vector and the amount of allowed play about this vector. When the orientation of the segment exceeds this amount of play, an external applied torque is applied to the segment in an attempt to right it. In Figure 7, the little man does not fall forward because the trunk is in the oriented state. At present, the strength of the orienting force is rather crudely determined by using a linear spring, so that while righting does occur it is likely to overshoot the mark and wobble or take too long. However, this may, with tuning, be useful in helping to stabilize the body during walking and standing. A extension of this technique to world space positions could allow the simulation of a body fixed about a point, such as hanging from a bar.

5. *Hybrid K-D Control*: Finally, while orienting and freezing simplify control, they do not directly address the problem of finding the controlling forces and torques for a specific motion. Because kinematic descriptions are far more convenient for the user, it is best to use positional motion specifications for those degrees of freedom under internal control and have the system determine the forces and torques necessary to achieve these positions. The use of positional instructions also provides a convenient interface to higher-level control routines whose output is likely to be in kinematic terms. This is related to the control problem in robotics, where the desired motion is known but the

forces and torques must be found.[16, 20, 17]

At present, *Deva* converts positional motion descriptions to forces and torques by calculating the velocity necessary to achieve the desired position in the upcoming time period, considering the mass of distal segments, and making an intelligent guess at the appropriate force or torque (see Appendix II). Because dynamics is done on a finer grain than the displayed animation, this simple method without feedback has proved acceptable. More sophisticated methods may be needed as more complex controlled motion is explored. Positional control is far more convenient than strictly dynamic control and generally gives equally realistic motion. Section 5 describes a sample animation session using hybrid control.

5. Virya: A Graphical Editor for Motion Control

To ease the problem of specifying control information for multiple-degree-of-freedom bodies, a graphical editor *Virya* has been implemented. *Virya* has two main functions. First, it allows the user to design and store controlling functions for each degree of freedom of the body. These functions may represent forces or torques over time when the degree of freedom is controlled directly by controlling forces and torques, or they may represent positions over time when the degree of freedom is controlled by position suggestions. Second, *Virya* allows the user to indicate the state of each degree of freedom over time, choices being those explained above (direct dynamic control, relaxed, frozen, oriented, or positional control). These functions are sampled each time dynamic analysis is done and used to determine the contribution of controlling forces and torques at each degree of freedom.

Virya operates on an Evans and Sutherland PS340 graphics system using a tablet and puck. Figure 8 shows the *Virya* screen. The lower half of the screen consists of a window describing the control function for each degree of freedom, as well as the name and type (x,y,z rotation or x,y,z translation) of the degree of freedom. The upper right quadrant shows the menu choices for interacting with *Virya*. The upper left quadrant is an expanded window to create and modify control functions, which are designed by designating defining points that specify a cubic interpolatory spline curve. Points can easily be added, moved, or deleted using the tablet. The control information can be stored in ASCII files.

Virya can also be used to specify and drive strictly kinematic animation. In this case, each control function represents only positions over time and animation is achieved by directly sampling the functions without recourse to dynamic analysis. *Deva* provides a 3-D keyframing facility where the user can specify a series of positions and associate a time with each. These position/time pairs for each degree of freedom become the defining points for the control function. Finally, because the output of the dynamic analysis routines is kinematic (consisting of a set of positions over time), it can be stored as kinematic control functions. This provides a succinct way to store the results of dynamic analysis and also allows kinematic tweaking of dynamically-predicted motion.

6. Sample Session

To illustrate the method, a brief session explaining how the animator can go from a keyframe file to a dynamically-predicted motion on the screen. A 24 degree-of-freedom body ("joe") will be used to create the motion of sitting up from a lying position on the floor.

1. Keyframing Approximate Motion: First, the user interactively positions the body in a selection of the positions to be interpolated and stores the time and position at each degree of freedom for each configuration. The keyframe positions are relevant for controlling the position of joints that are *frozen* in their local position or under *hybrid K-D* control. Joints that are *relaxed* will move freely from their initial positions due to the environment. *Oriented* degrees of freedom will attempt to maintain their world space configuration. *Dynamically-controlled* joints would act according to user-specified forces and torques. Oriented and dynamically-controlled joints are not present in this example. Figure 9 shows the initial keyframe file for this motion. Figure 10 shows the body as defined by this file. Note that the positions of some body parts (such as the arms) are unnatural. These joints will be relaxed in the actual motion and dynamic analysis will ensure they appear in a natural position.

2. Keyframe to Virya File Conversion: Next, the user asks *Deva* to convert the keyframe file to a Virya input file. In this process, data is rearranged so that positions are associated with their degree of freedom rather than the time when they occur. Using keyframing, all degrees of freedom are by default assumed to be in hybrid K-D mode. If run, the system would attempt to achieve the specified position for each degree of freedom. This, however, would likely be counterproductive, since dynamics is unlikely to influence the motion in any considerable way (besides possibly indicating that the desired positions are unreachable).

3. Modification of the Virya Input File: To take advantage of the dynamics, the user can then use Virya to modify the input file into a form where various degrees of freedom are under different modes of control. In the case of sitting up, the hybrid mode is necessary only at the waist and neck. The hips are frozen so that the body lifts its trunk not its legs when a torque is applied at the waist. The rest of the joints are in relaxed mode and lie naturally in reference to the floor. Figure 11 shows the Virya input file after modification.

4. Dynamic Analysis: The Virya input file is now sent to Deva and dynamic analysis is used to predict motion. Analysis is done 300 times per second. The user can specify that some or all of these configurations are sent to the display routines or can save the output positions in a kind of output keyframe file indicating the times and positions predicted for dynamic analysis.

5. Output Keyframe File to Virya Output File The user asks Deva to convert the output keyframe file to a Virya file (again arranging positions by degree of freedom not time). This file now can be used to generate a cubic spline curve describing smooth motion at each degree of freedom. Using Deva in its strictly kinematic mode, the file is sampled to produce output at any desired sample rate. Figure 12 shows a sequence of positions generated from the output Virya file.

7. Problems with Dynamic Motion Control and Possible Solutions

The three problems that most hamper dynamic motion control are computational complexity, numerical instability, and motion control.

7.1. Computational Cost

Using the slow Gibbs-Appell dynamics, only a few seconds of animation are calculated per hour (on a VAX-750); however, the Armstrong formulation is close enough to realtime to be interactive for a patient user (on a 68020-based workstation).[12, 19] Armstrong, Green, and Lake have suggested used parallel processing for even great speed.[12] Speed is affected not only by the basic dynamics, but by the amount of control required, particularly in the case of collision detection and response.

Speed could also be increased by using dynamic analysis on a subset of the body segments. (The motion of light body segments such as the hands contribute minimally to the total dynamic interactions of the whole body.) The motion suggested by the dynamics routines could then be used to partially control the motion of a much more complicated body, leaving the user to determine kinematically the motion at additional degrees of freedom. Alternatively, the use of dynamics can be severely simplified to the minimal analysis needed to give some feeling of "mass" to the motion. Girard and Maciejewski[37] have used a simple version of dynamic analysis that is applied only to the trunk segment of their articulated bodies.

7.2. Numerical Instability

The second problem is numerical instability. In the present dynamics formulation, considerable oversampling must done to ensure solution of the equations; i.e., the dynamics equations are formulated and solved from 50 to 300 times per second, while video recording occurs 30 times per second. The problem is most serious when the body has many degrees of freedom and the motion is very controlled. If sampling is insufficiently frequent, a solution cannot be found for the dynamics equations and the program dies. Adaptive sampling and the ability to automatically restart the program when this occurs would be helpful. Armstrong artificially increases the moments of inertia around longitudinal axes to minimize the problem,[11] but this can interfere with the realism of the movement.

7.3. Motion Control

The third and most serious problem is motion control. While motion dominated by effects from the external environment such as gravity is automatically calculated, some reliable, convenient, and kinematically-based method must be found to allow the user to specify motion under internal "pseudo-muscular" control. There are two problems: first, on a low-level, how to achieve the desired positions at controlled joints; and, second, on a higher-level, how to coordinate complex motion.[5]

To consider the low-level problem, assume the desired local joint positions for part of the body are known (how is another issue, whether from user interaction or inverse kinematics[30, 19]), and the torques (assuming revolute joints) must be specified. Using robotic control methods, direct dynamics could be applied to this subset of the body to find controlling torques, which can be used with indirect dynamics on the entire body to find total body motion.[16] This may not work: e.g., consider a person turning quickly in response to a sudden sound. The user knows the motion of the trunk and neck, but would like a realistic relaxed motion of the arms. If the torques for the trunk rotation are calculated without considering the arms, they will not produce the expected motion when used in indirect dynamics, because the motion of the arms will also affect the trunk and neck.

If the arm positions were already known, there would be no point in dynamics. Recursive dynamics formulations assume one is solving either for forces and torques or accelerations.[20, 17] It might be possible to reformulate the equations in such a way that these can be combined; this should be explored.

An alternative is to calculate desired torques on the fly during the dynamics, as is done in hybrid control here. In the example, a waist torque could be estimated given the body mass and desired motion, and altered appropriately over time depending on whether the calculated motion is faster or slower than that needed. Various feedback schemes are possible and should be tested for accuracy versus speed in this application.[38]

The second issue is higher-level control for coordinated motion. A simple example might illustrate the problem. Consider modeling a horse rearing on its hind legs and then landing on all fours. When on two legs, the hindquarters must apply sufficient torques to support the upper body, and alter these torques to compensate for the pawing motion of the front legs. When the front legs fall, due to relaxation of the back leg torques, their joints will be fairly relaxed. But as soon as the front legs strike the ground, their joints must stiffen to accept the body weight; otherwise, they will splay out and the horse will fall on its face. The user would not know at exactly what time the legs must tense, so control must be specified in terms of "when this event occurs do this".[5]

Considerable work has been done in robotics and control theory dealing with these issues, and this literature should be exploited in future work. However, though related, the problems involved are not identical. In one sense, the animation problem is more difficult, as most robots are far simpler than the articulated bodies shown in computer graphics. Although some excellent progress has been made in creating bipedal robots,[39] two-legged walking robots capable of the complex fluid movement we associate with normal human motion have yet to be built. But in another, the animation problem is simpler, because only the *appearance* of realism is needed, not actual physical reality.

8. Conclusions

Dynamically-controlled animation offers the possibility of adding a new level of realism to worlds created by computer animation. Use of dynamics naturally restricts motion to realizable patterns and automatically calculates the effects of many environmental interactions which are very difficult to deal with kinematically. Though a number of difficult problems exist, preliminary results suggest the method has great potential.

Appendix I: Details of the Gibbs-Appell Dynamics Formulation

I.1. Derivation of the Gibbs-Appell Formulation

The Gibbs-Appell dynamics formulation is based on the Gibbs Formula, which describes the *energy of acceleration.*[29] For rigid bodies consisting of n segments, this formula is

$$G = \sum_{k=1}^{n} \left(\frac{1}{2} m_k \mathbf{a}_k^T \mathbf{a}_k + \frac{1}{2} \alpha_k^T \mathbf{I}_k \alpha_k + \alpha_k^T (\omega_k \times I_k \omega_k) + f(\omega_k) \right)$$

where, for segment k,

m_k = *mass*

$\mathbf{a}_k$ = *acceleration vector of center of mass*

α_k = *angular acceleration vector*

ω_k = *angular velocity vector*

$f(\omega_k)$= *scalar disappears with differentiation*

$\mathbf{I}_k = \begin{bmatrix} I_{xx} I_{xy} I_{xz} \\ I_{xy} I_{yy} I_{yz} \\ I_{xz} I_{yz} I_{zz} \end{bmatrix}$ = *inertial tensor*

$I_{xx} = \int (y^2 + z^2) dm$; *etc., moments of inertia*

$I_{xy} = \int xy\ dm$; *etc., products of inertia*

The dynamics equations given in Section 2.1 are found by partially differentiating the Gibbs formula with respect to the local acceleration relative to each degree of freedom.

I.2. Calculating the Terms of the Dynamics Equations

Explanation of Terms: The partial differentiation of the Gibbs formula leaves motion described in inertial world space terms; however, these equations can be restated in terms of local joint configuration because of the known relation between local and world frames. Together with the terms described in Section 2.1, the following kinematic configuration information is necessary ($k = 1, \ldots, n$ for n degrees of freedom).

$\mathbf{r}_k$ = *3-D position vector of joint connecting segments* $k-1$ and k

$\mathbf{g}_k$ = *3-D position vector of center of mass of segment* k

$\mathbf{u}_k$ = *3-D unit vector of rotation or sliding axis of joint* k

$s_k, \dot{s}_k, \ddot{s}_k$ = *sliding along axis* $\mathbf{u}_k$, its velocity, and acceleration

$\theta_k, \dot{\theta}_k, \ddot{\theta}_k$ = *rotation angle about axis* $\mathbf{u}_k$, its angular velocity, and acceleration

$\mathbf{R}_k$ = 3×3 matrix describing rotation about axis $\mathbf{u}_k$

Finding the Inertial Matrix (M-Matrix): $\mathbf{M}$ is an $n \times n$ matrix which takes into account the present distribution of body mass. $\mathbf{M}$ consists of four submatrices.

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}^{\theta} & \mathbf{M}^{\theta s} \\ [\mathbf{M}^{\theta s}]^T & \mathbf{M}^{s} \end{bmatrix}$$

The upper left submatrix $\mathbf{M}^{\theta}$ is an $r \times r$ matrix describing the relation between revolute degrees of freedom. Its elements are defined by the following equation

$$m_{ij}^{\theta} = m_{ji}^{\theta} = \sum_{k=distal(i,j)}^{Todistal} (m_k [\mathbf{u}_i \times (\mathbf{g}_k - \mathbf{r}_i)]^T$$
$$[\mathbf{u}_j \times (\mathbf{g}_k - \mathbf{r}_j)] + \mathbf{u}_i^T \mathbf{I}_k \mathbf{u}_j)$$

$$(\text{for } i = 1, \ldots, r \text{ and } j = 1, \ldots, r).$$

(Note that *distal* (i,j) refers to whichever of i or j lies further from the initial world segment, and if i and j lie on separate branches the calculation does not take place. *Todistal* refers to the furthest segments continuing out this branch.)

The upper right submatrix $\mathbf{M}^{\theta s}$ (whose transpose is the lower left submatrix) is an $r \times t$ matrix describing the relation between revolute and sliding degrees of freedom. Its elements are

$$m_{ij}^{\theta s} = \sum_{k=distal(i,j)}^{Todistal} m_k \mathbf{u}_j^T [\mathbf{u}_i \times (\mathbf{g}_k - \mathbf{r}_i)]$$

$$(\text{for } i = 1,..,r \text{ and } j = 1,..,t).$$

The lower right submatrix $\mathbf{M}^{s}$ is a $t \times t$ matrix describing the relation between sliding degrees of freedom. Its elements are

$$m_{ij}^{s} = m_{ji}^{s} = \sum_{k=(i,j)}^{Todistal} m_k \mathbf{u}_i^T \mathbf{u}_j$$

(for $i = 1, \ldots, t$ and $j = 1, \ldots, t$).

Velocity-Dependent V-Vector: The **V**-vector takes into account such velocity-dependent contributions as the Coriolis and centrifugal forces. The **V**-vector contains two sub-vectors: $\mathbf{V}_\theta$ is an r-length vector representing revolute degrees of freedom, and $\mathbf{V}_s$ is a t-length vector representing sliding degrees of freedom.

$$\mathbf{V} = \begin{bmatrix} \mathbf{V}_\theta \\ \mathbf{V}_s \end{bmatrix} \quad \mathbf{V}_\theta = \begin{bmatrix} V_{\theta 1} \\ \cdots \\ V_{\theta r} \end{bmatrix} \quad \mathbf{V}_s = \begin{bmatrix} V_{s1} \\ \cdots \\ V_{st} \end{bmatrix}$$

The elements of the $\mathbf{V}_\theta$ vector for revolute degrees of freedom ($V_{\theta k}$ for $k = 1, \ldots, r$) are found using the following equation.

$$V_{\theta k}=[\dot\theta_1,..,\dot\theta_r,\dot s_1,..,\dot s_t]\begin{bmatrix} \mathbf{N}_\theta^{k\theta} & \mathbf{N}_\theta^{ks} \\ [\mathbf{N}_\theta^{ks}]^T & 0 \end{bmatrix}[\dot\theta_1,..,\dot\theta_r,\dot s_1,..,\dot s_t]^{\mathrm{T}}$$

where the components of the $\mathbf{N}_\theta$ matrix are found by

$$n_{\theta ij}^{k\theta}=n_{\theta ji}^{k\theta}=\sum_{l=distal(k,j)}^{distal}(m_l[\mathbf{u}_k\times(\mathbf{g}_l-\mathbf{r}_k)]^T[\mathbf{u}_i\times(\mathbf{u}_j\times(\mathbf{g}_l-\mathbf{r}_j))]$$
$$+\mathbf{u}_k^T[\frac{1}{2}trace(\mathbf{I}_l)\mathbf{u}_i\times\mathbf{u}_j+\mathbf{u}_j\times\mathbf{I}_l\mathbf{u}_i])$$

where $trace(\mathbf{I}_l) = I_{l11}+I_{l22}+I_{l33}$

(for $i = 1, \ldots, r$ and $j = 1, \ldots, r$)

and

$$n_{\theta ij}^{ks}=\begin{cases} \displaystyle\sum_{l=distal(k,j)}^{distal} m_l[\mathbf{u}_k\times(\mathbf{g}_l-\mathbf{r}_k)]^T[\mathbf{u}_i\times\mathbf{u}_j] & \textit{for i proximal to j} \\ 0 & \textit{for i distal to j} \end{cases}$$

(for $i = 1, \ldots, r$ and $j = 1, \ldots, t$)

The elements of $\mathbf{V}_s$ vector for sliding degrees of freedom (V_{sk} for $k = 1, \ldots, t$) are found using the following equation.

$$V_{sk}=[\dot\theta_1,..,\dot\theta_r,\dot s_1,..,\dot s_t]\begin{bmatrix} \mathbf{N}_s^{k\theta} & \mathbf{N}_s^{ks} \\ [\mathbf{N}_s^{ks}]^T & 0 \end{bmatrix}[\dot\theta_1,..,\dot\theta_r,\dot s_1,..,\dot s_t]^{\mathrm{T}}$$

where the components of the $\mathbf{N}_s$ matrix are found by

$$n_{sij}^{k\theta}=n_{sji}^{k\theta}=\sum_{l=distal(k,j)}^{distal} m_l \mathbf{u}_k^T[\mathbf{u}_i\times(\mathbf{u}_j\times(\mathbf{g}_l-\mathbf{r}_j))]$$

$$(\text{for } i = 1, \ldots, r \text{ and } j = i, \ldots, r\ )$$

and

$$n_{sij}^{ks} = \begin{cases} \sum_{l=distal(k,j)}^{distal} m_l \mathbf{u}_k^T[\mathbf{u}_i\times\mathbf{u}_j] & i \textit{ proximal to } j \\ 0 & i \textit{ distal to } j \end{cases}$$

$$(\text{for } i = 1, \ldots, r \text{ and } j = 1, \ldots, t)$$

Appendix II: Calculation of the Forces and Torques Contributing to Motion

II.1. Gravitational Forces and Torques

The gravitational component of each generalized force can be determined simply by considering the effect of the mass of distal segments at a particular degree of freedom. In the case of each revolute degree of freedom, this involves finding the torque around its axis of rotation, which depends upon the gravitational force acting on each distal segment and the perpendicular distance of the point of force's application from the axis of rotation. The equation for torque due to gravity at degree of freedom k is

$$\tau_{gk} = -g_c \sum_{i=k}^{distal} m_i \mathbf{z}_{0T} [\mathbf{u}_k \times (\mathbf{g}_i - \mathbf{r}_k)]$$

where $\mathbf{z}_0 = (0,0,1)$ is a vertical direction vector, $g_c = 9.81$ m/sec^2 is the acceleration due to gravity (on the earth's surface), and the other terms are as explained previously.

In the case of each sliding degree of freedom, the gravitational component (f_{gk}) contributing to the generalized force there is dependent upon the component of the gravitational force acting on each distal segment that lies along the axis of sliding; that is,

$$f_{gk} = -g_c \sum_{i=k}^{distal} m_i \mathbf{z}_0^T \mathbf{u}_k$$

II.2. External Applied Forces

Where the magnitude and direction of an external applied force (such as a pull from a rope or a shove) is known, the contribution of this force to the generalized force at each degree of freedom can be calculated using a method very similar to that used for gravity. Assume that a force represented by vector $\mathbf{F}_a$ is applied to segment a at a location designated by the world-space vector $\mathbf{a}$. Each revolute degree of freedom i which lies proximal to segment a feels the effect of this applied force as the torque τ_{app_i} defined by the equation

$$\tau_{app\_i} = \mathbf{F}_a [\mathbf{u}_i \times (\mathbf{a} - \mathbf{r}_i)]$$

Each sliding degree of freedom j which lies proximal to segment a feels the effect of the applied force as the force F_{app_j} defined by the equation

$$F_{app\_j} = \mathbf{F}_a \mathbf{u}_j$$

II.3. Joint Limit Forces and Torques and Damping

The joint limit forces (for sliding joints) or torques (for revolute joints) are each simulated using three components. The first component counteracts other forces and torques pushing the joint beyond its limit and is simply equal in magnitude and opposite in direction to all other forces (for sliding joints) or torques (for revolute joints) contributing to the generalized force at that degree of freedom (such as gravity or actuator forces). The second component is a spring whose strength is a function of the amount the joint limit has been exceeded, the local velocity, and the mass (for sliding joints) or moment of inertia (for revolute ones) distal to this degree of freedom. The third component is a damper which is a function of the local velocity and the mass or moment of inertia due to distal segments. The reasoning behind the method used to calculate these forces and torques is rather complex and explained fully elsewhere.[14] This method is superior to a spring and damper proportional only to the amount of compression and the velocity because it automatically adjusts for differences in mass of different segments. Without such a consideration, individual spring and dampers constants would have to be developed for different joints, a considerable burden to the user.

The spring force f_{spr_jl} and the damping force f_{dmp_jl} opposing motion beyond the joint limit for sliding joints are

$$f_{spr\_jl} = -k_{spr\_sl} \times m\_dist \times vel\_loc \times delx$$

$$f_{dmp\_jl} = -k_{dmp\_sl} \times m\_dist \times vel\_loc$$

where m_dist is the sum of all masses distal to this degree of freedom, vel_loc is the local velocity at this degree of freedom, $delx$ is the amount the joint limit has been exceeded, k_{spr_sl} and k_{dmp_sl} are constants affecting how strongly motion is opposed (typically $k_{spr\_jl} = 1000\frac{m}{\text{sec}}$ and $k_{dmp\_jl} = 1.4\frac{1}{\text{sec}}$).

The spring torque τ_{spr_jl} and the damping torque τ_{dmp_jl} for revolute joints are calculated in a similar fashion, with m_dist being the moment of inertia at this degree of freedom due to distal masses, vel_loc being an angular velocity, $delx$ being an angular distance, and the constants being typically $k_{spr\_rl} = 55\frac{rad}{\text{sec}}$ and $k_{dmp\_rl} = 1.4\frac{1}{\text{sec}}$.

A damping action is normally active throughout the range of joint motion. This damper is calculated in the same manner as the end limit damper but is weaker (usually 60% of end damping).

II.4. Ground Reaction Forces

Simulating ground reaction forces with an approximated force plus springs and dampers has been found a satisfactory method. Forces are applied at any of the eight corners of a max-min box surrounding each body segment which are on or below the level of the ground. Reaction forces consist of a normal force perpendicular to the ground and two orthogonal tangential forces.

Calculating Normal Forces: Normal forces are calculated as a combination of an estimated reaction force plus, possibly, a contribution from a spring and damper. The estimated total reaction force for the whole body (taking into account its total mass and approximated total momentum) is

$$f_{rf\_tot} = -g \times m\_tot - k\_rf \times m\_tot \times v\_root$$

where g is the acceleration due to gravity, m_tot is the total body mass, v_root is the vertical velocity of the root segment of the body (the trunk or abdomen), and k_rf is a constant (typically $.5\frac{\text{sec}}{m}$). This reaction force is distributed between all contact points depending upon the relative depth of the contact points below the floor. For example, if the sum of the distances below the floor of all contact points were 1 centimeter, and a particular contact point were .1 centimeter into the floor, it would receive .1/1 or 10% of the reaction force.

If the contact point continues to descend into the floor, this estimated local force is increased by a factor of $.2 \times \delta z$ where δz is the distance of further descent in meters.

Calculating Tangential Frictional Forces: Frictional forces act to oppose tangential motion along the surface and are dependent upon both the normal force pressing into the ground and the characteristics of the surfaces. To simulate frictional forces, a combination of an estimated force plus a spring and damper are again used.

The estimated tangential force for a particular contact point is merely the product of a coefficient of friction and the normal force calculated as indicated above and applied in a direction to oppose sliding. If the contact continues to be displaced tangentially, a tangential spring and damper can be applied to contribute further opposition.

The reaction force is then treated as an external applied force acting on the contact point, as described earlier in this appendix. The effect of reaction forces can be altered by varying the constants described above to simulate more or less springiness, damping, and friction; e.g. bodies sliding on ice or bouncing on trampolines.

II.5. Controlling Forces and Torques

At those degrees of freedom whose motion is being explicitly controlled by the user, controlling forces and torques must be input as control functions describing the force or torque versus time for each degree of freedom (the direct method) or describing the desired positions over time for each degree of freedom (the indirect method). The indirect method is more intuititive.

In the indirect method, the control function is sampled to find the desired local position at the next time sample. A trivially simple method is used to convert this information to a controlling force or torque.

$$ft_{hkd} = m\_dist \times \left(\frac{\delta\_pos}{\delta\_time} - vel\right) \times \frac{1}{\delta\_time}$$

where *m_dist* is the sum of the masses (sliding joints) or moments of inertia (revolute joints) distal to this degree of freedom, δ*_time* is the time between time samples, δ*_pos* is the distance between the desired next position and the present position, and *vel* is the present velocity. This method is successful in the present case despite its crudeness because dynamic analysis is typically done much more often than imaging, so actual animated motion appears smooth.

Figure 1. Forces and Torques Contributing to Motion

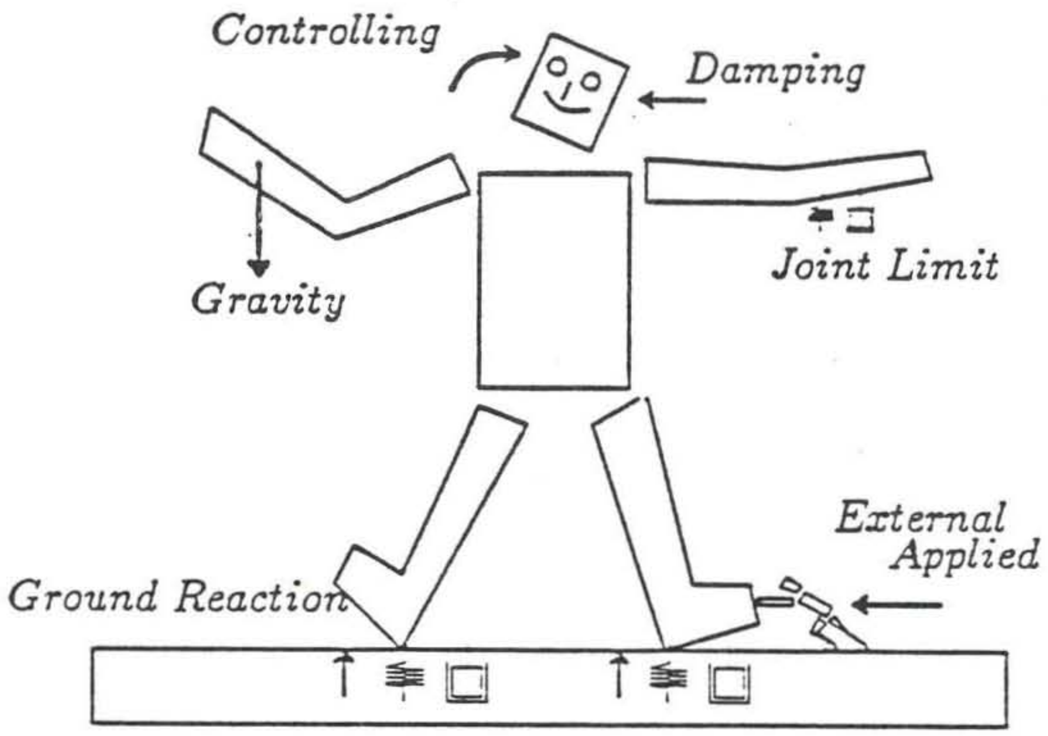


Figure 2. Motion due to Gravity Without Restraining Ground Forces

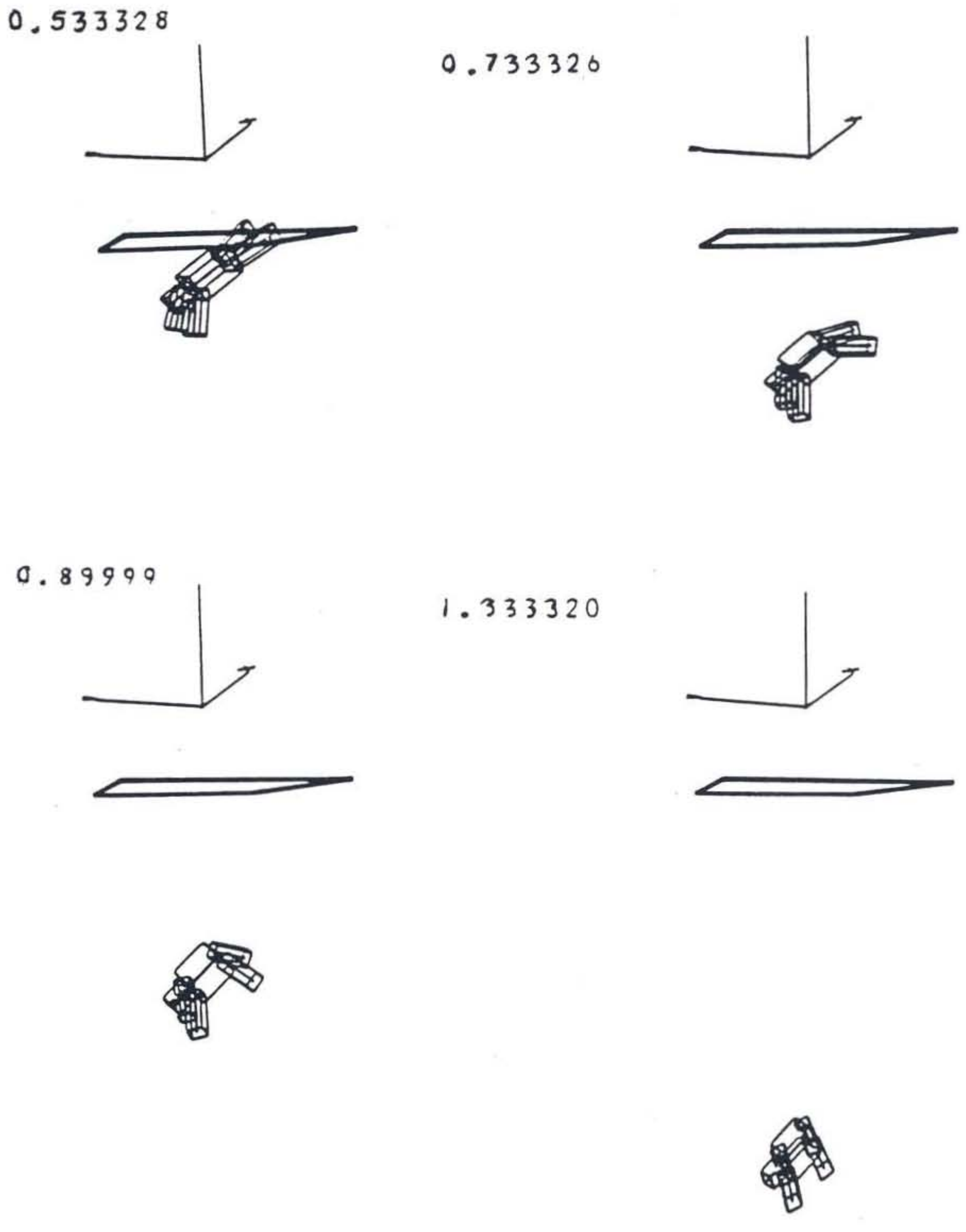


Figure 3. Motion due to Gravity Without Joint Limits

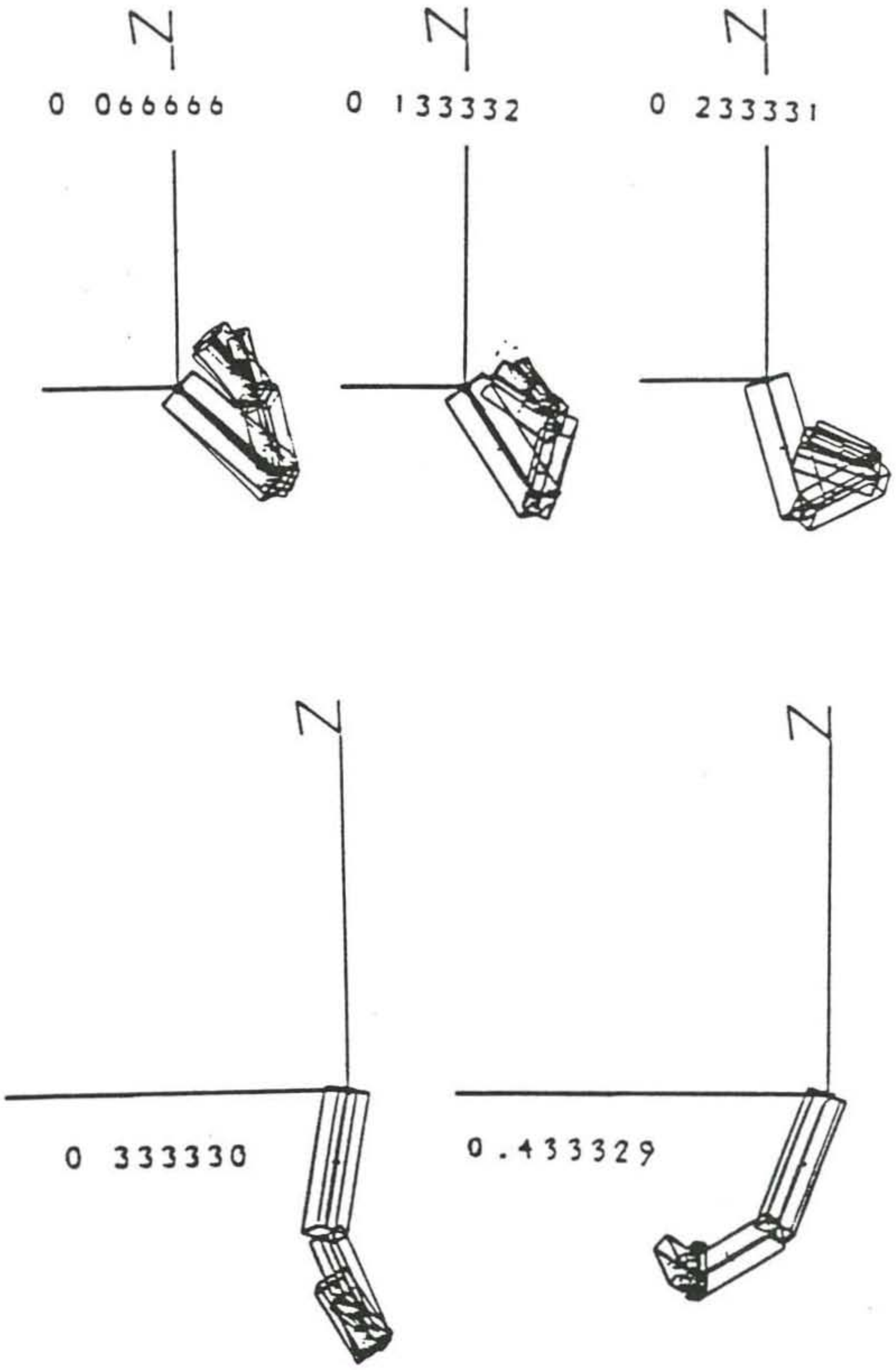


Figure 4. Controlled Motion of the Arm

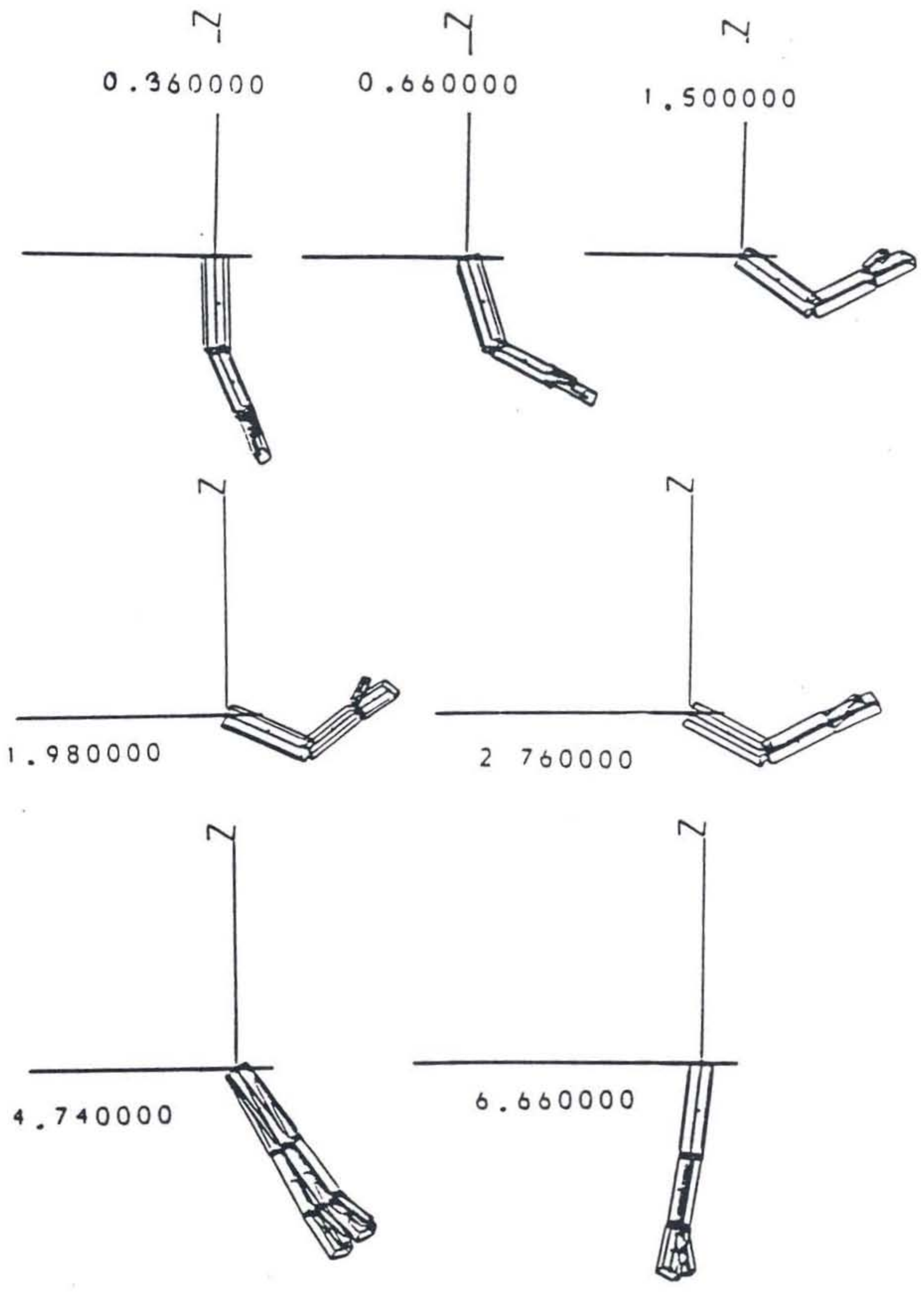


Figure 5. Falling onto a Slippery Ground

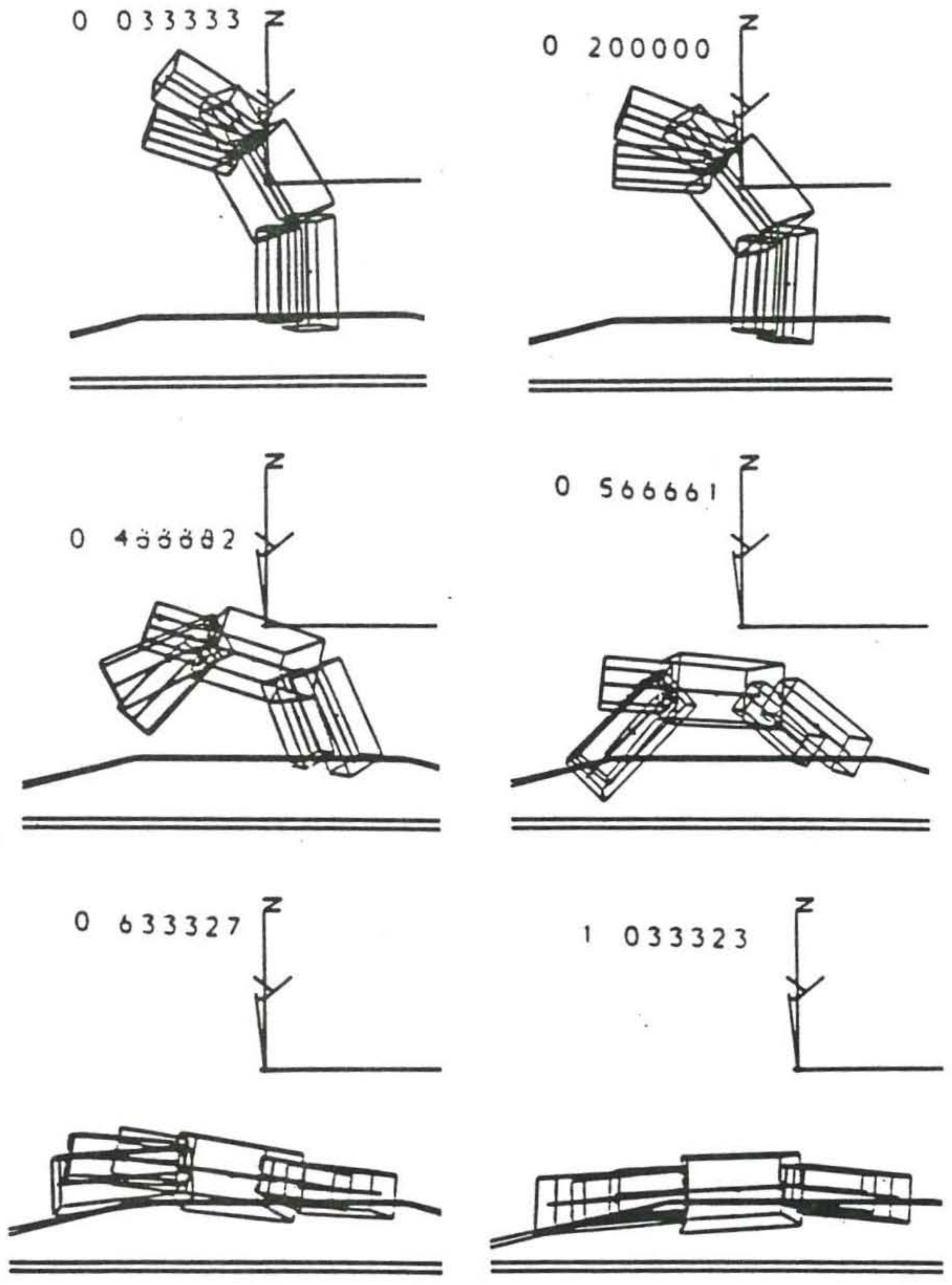


Figure 6. Falling onto a Sticky Ground

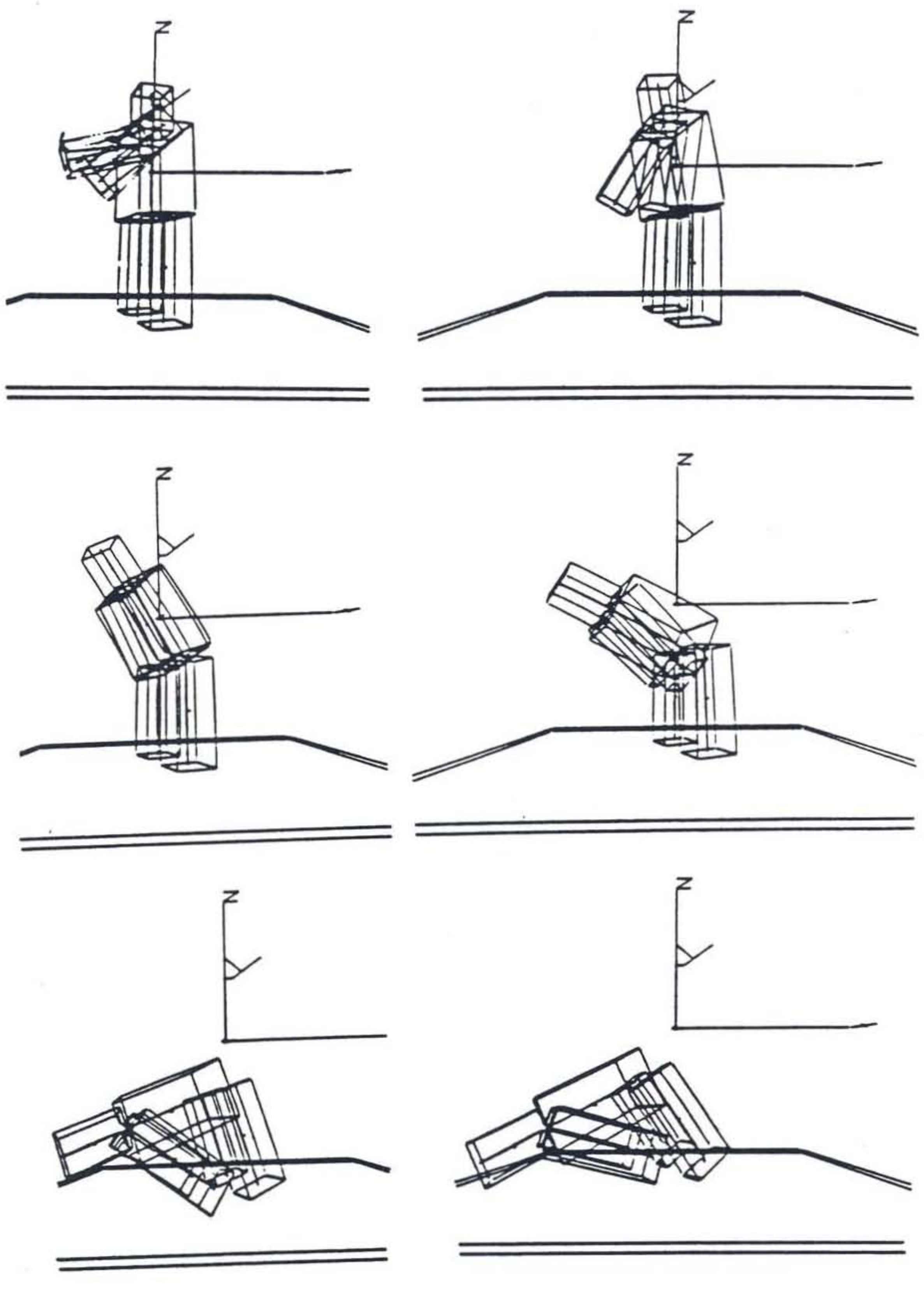


Figure 7. Balance Applied to the Torso

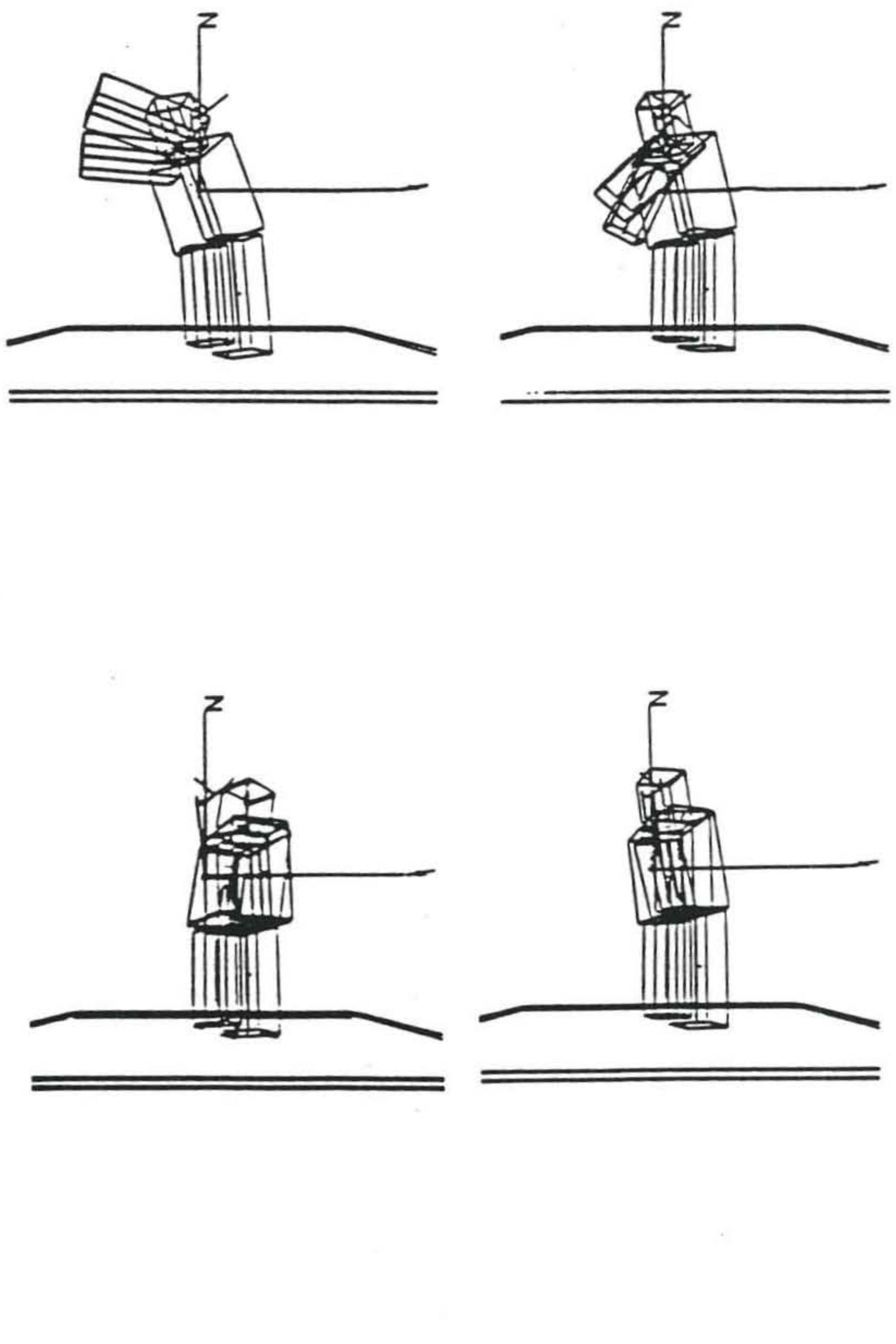
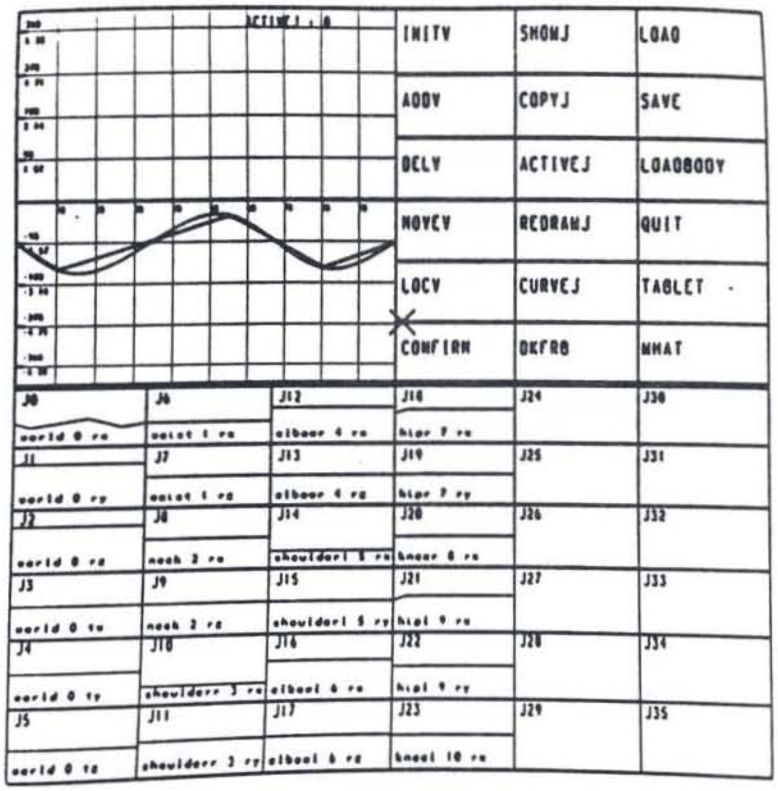


Figure 8. The Virya Screen



This page intentionally
left blank.

Figure 9. Initial Input Keyframe File for Sitting Up

```
0.000000 /* time zero */
-90.000000 180.000000 180.000000 0.000000 0.000000 -0.930000 0.000000
0.000000 0.000000 0.000000 -180.000000 0.000000 0.000000 0.000000
-180.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
2.000000 /* time 2 seconds */
-90.000000 180.000000 180.000000 0.000000 0.000000 -0.930000 -45.000000
0.000000 0.000000 0.000000 -180.000000 0.000000 0.000000 0.000000
-180.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
4.000000 /* time 4 seconds */
-90.000000 180.000000 180.000000 0.000000 0.000000 -0.930000 -90.000000
0.000000 0.000000 0.000000 -180.000000 0.000000 0.000000 0.000000
-180.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
5.000000 /* time 5 seconds */
-90.000000 180.000000 180.000000 0.000000 0.000000 -0.930000 -90.000000
0.000000 0.000000 0.000000 -180.000000 0.000000 0.000000 0.000000
-180.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
100.000000 /* time 100 seconds */
-90.000000 180.000000 180.000000 0.000000 0.000000 -0.930000 -90.000000
0.000000 0.000000 0.000000 -180.000000 0.000000 0.000000 0.000000
-180.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
```

Figure 10. Keyframe Configurations Displayed

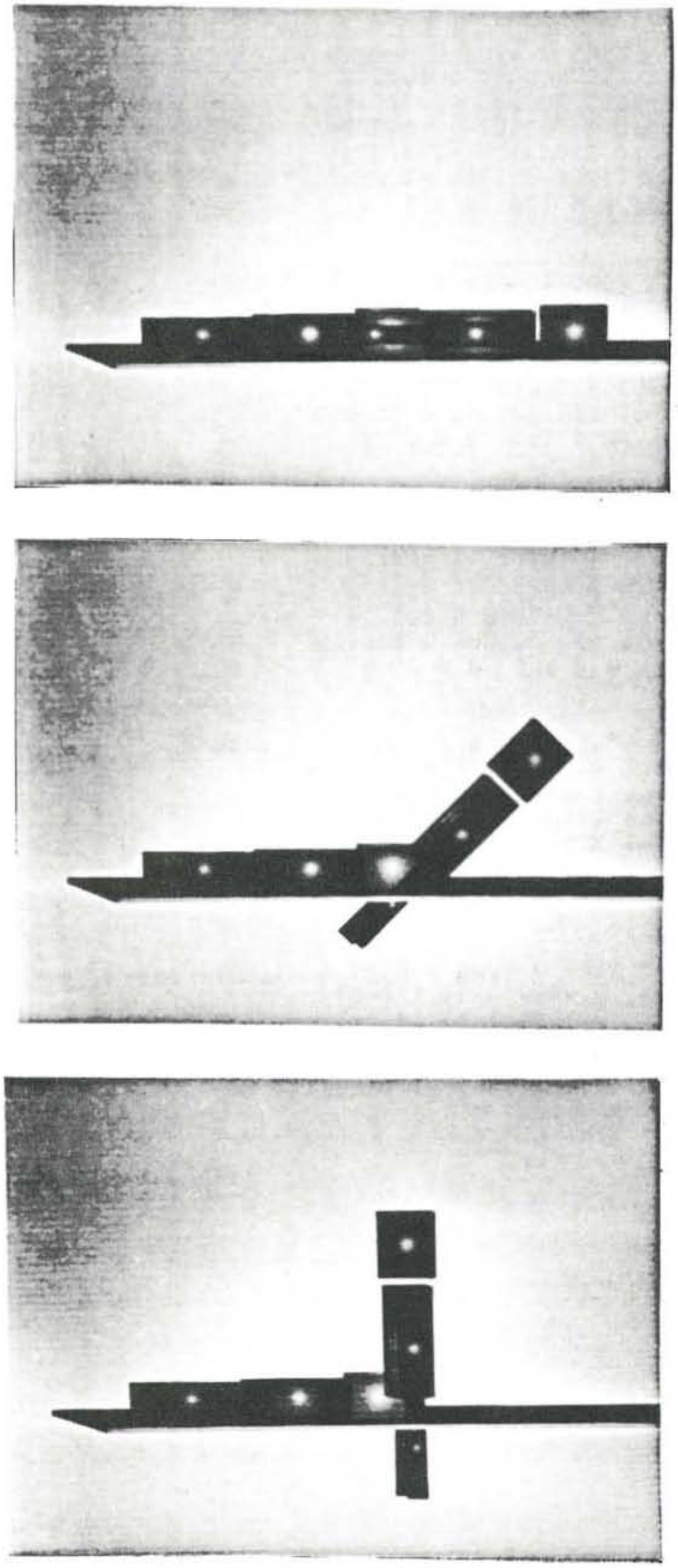
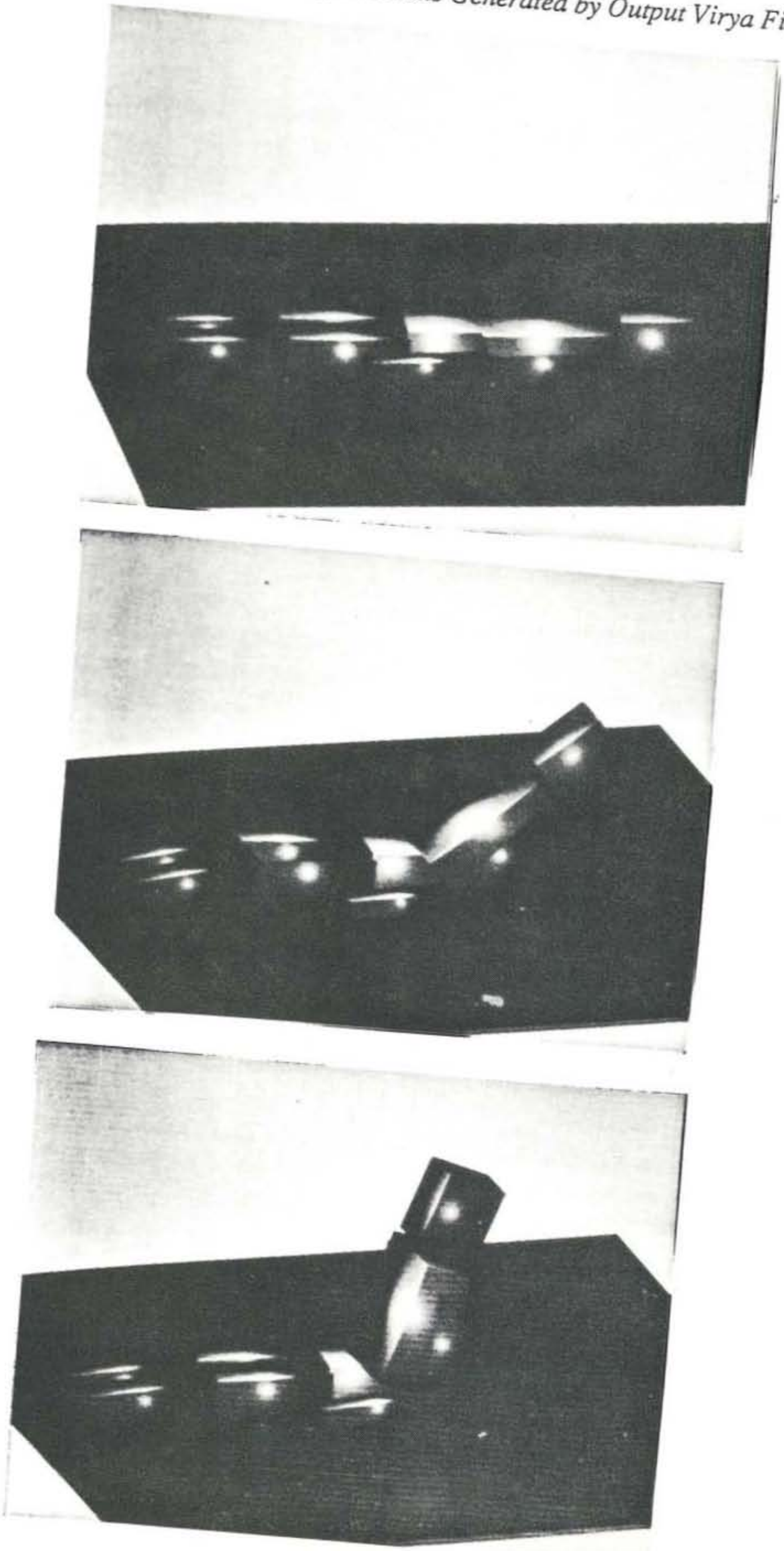


Figure 11   Input Virya File Modified for Dynamics (Partial)

```
dof 6 Pjnt 1 Type 0 Numv 5  j_waist
Control 0.000000 0.000000
Control 2.000000 -0.785398
Control 4.000000 -1.570796
Control 5.000000 -1.570796
Control 100.000000 -1.570796
States K 0 100  0 0 0  0
Q

dof 7 Pjnt 1 Type 2 Numv 5  j_waist
Control 0.000000 0.000000
Control 2.000000 0.000000
Control 4.000000 0.000000
Control 5.000000 0.000000
Control 100.000000 0.000000
States F 0 100  0 0 0  0
Q

dof 8 Pjnt 2 Type 0 Numv 5  j_neck
Control 0.000000 0.000000
Control 2.000000 0.000000
Control 4.000000 0.000000
Control 5.000000 0.000000
Control 100.000000 0.000000
States F 0 100  0 0 0  0
Q

dof 9 Pjnt 2 Type 2 Numv 5  j_neck
Control 0.000000 0.000000
Control 2.000000 0.000000
Control 4.000000 0.000000
Control 5.000000 0.000000
Control 100.000000 0.000000
States F 0 100  0 0 0  0
Q

dof 10 Pjnt 3 Type 0 Numv 5  j_shoulderr
Control 0.000000 -3.141593
Control 2.000000 -3.141593
Control 4.000000 -3.141593
Control 5.000000 -3.141593
Control 100.000000 -3.141593
States R 0 100  0 0 0  0
Q
```

Figure 12. Sequence of Positions Generated by Output Virya File



References

1. Norman I. Badler and Stephen W. Smoliar, ''Digital Representation of Human Movement,'' *ACM Computing Surveys*, vol. 11, pp. 19-38, March, 1979.
2. Thomas W. Calvert, J. Chapman, and A. Patla, ''Aspects of the Kinematic Simulation of Human Movement,'' *IEEE Computer Graphics and Applications*, vol. 2, no. 9, pp. 41-52, November, 1982.
3. Marianne Dooley, ''Anthropometric Modeling Programs - A Survey,'' *IEEE Computer Graphics and Applications*, vol. 2, no. 9, pp. 17-26, November, 1982.
4. Don Herbison-Evans, ''Nudes 2: A Numeric Utility Displaying Ellipsoid Solids,'' *SIGGRAPH '78 Conference Proceedings*, pp. 354-356, August, 1978.
5. David Zeltzer, ''Motor Control Techniques for Figure Animation,'' *IEEE Computer Graphics and Applications*, vol. 2, no. 9, pp. 53-60, November, 1982.
6. Dare A. Wells, *Lagrangian Dynamics*, Shaum's Outline Series, McGraw-Hill Book Co., New York, 1969.
7. Scott N. Steketee and Norman I. Badler, ''Parametric Keyframe Interpolation Incorporating Kinetic Adjustment and Phrasing Control,'' *SIGGRAPH '85 Conference Proceedings*, pp. 255-262, July, 1985.
8. David Sturman, ''Interactive Keyframe Animation of 3-D Articulated Models,'' *SIGGRAPH '85 Tutorial Notes: Introduction to Computer Animation*, July, 1985.
9. Craig W. Reynolds, ''Description and Control of Time and Dynamics in Computer Animation,'' *SIGGRAPH '85 Tutorial Notes: Advanced Computer Animation*, pp. 289-296, July, 1985.
10. K. D. Willmert, ''Visualizing Human Body Motion Simulations,'' *IEEE Computer Graphics and Applications*, vol. 2, no. 9, pp. 35-43, November, 1982.
11. William W. Armstrong and Mark W. Green, ''The Dynamics of Articulated Rigid Bodies for Purposes of Animation,'' *Proceedings of Graphics Interface '85*, pp. 407-415, Computer Graphics Society, May, 1985.
12. William W. Armstrong, Mark Green, and R. Lake, *Proceedings of Graphics Interface 86*, pp. 147-151, May, 1986.
13. Jane Wilhelms and Brian A. Barsky, ''Using Dynamic Analysis for the Animation of Articulated Bodies such as Humans and Robots,'' *Proceedings of Graphics Interface '85*, pp. 97-104, May 1985.
14. Jane Wilhelms, ''Graphical Simulation of the Motion of Articulated Bodies such as Humans and Robots, with Particular Emphasis on the Use of Dynamic Analysis,'' *PhD Thesis*, Computer Science Division, Berkeley, CA, July, 1985.
15. R. Featherstone, ''The Calculation of Robot Dynamics Using Articulated-Body Inertias,'' *International Journal of Robotics Research*, vol. 2, no. 1, pp. 13-30, Spring, 1983.
16. C. S. George Lee, R. C. Gonzalez, and K. S. Fu, *Tutorial on Robotics*, IEEE Computer Society Press, Silver Spring, MD, 1983.
17. M. W. Walker and D. E. Orin, ''Efficient Dynamic Computer Simulation of Robotic Mechanisms,'' *Journal of Dynamic Systems, Measurement, and Control*, vol. 104,

pp. 205-211, September, 1982.

18. Daniel E. Whitney, ''Resolved Motion Rate Control of Manipulators and Human Prostheses,'' *IEEE Transactions on Man-Machine Systems*, vol. MMS-10, no. 2, pp. 47-53, June, 1969.

19. Jane Wilhelms, David Forsey, and Pat Hanrahan, *Manikin: Dynamic Analysis for Articulated Body Manipulation*, Computer and Information Sciences Board, UCSC, Santa Cruz, CA , April, 1987. Technical Report UCSC-CRL-87-2

20. Richard P. Paul, *Robot Manipulators: Mathematics, Programming, and Control,* The MIT Press, Cambridge, MA, 1981.

21. Hooshang Hemami, ''Modeling, Control, and Simulation of Human Movement,'' *CRC Critical Reviews in Biomedical Engineering*, vol. 13, no. 1, pp. 1-34, 1985.

22. Miles Townsend and Ali Seireg, ''Effect of Model Complexity and Gait Criteria on the Synthesis of Bipedal Locomotion,'' *IEEE Transactions on Biomedical Engineering*, vol. BME-20, no. 6, November, 1973.

23. R. J. Williams and Ali Seireg, ''Interactive Computer Modeling of the Musculoskeletal System,'' *IEEE Transactions on Biomedical Engineering*, vol. BME-24, no. 3, pp. 213-219, May, 1977.

24. Doris I. Miller, ''Computer Simulation of Human Motion,'' in *Techniques for the Analysis of Human Motion*, ed. D. W. Grieve et al., Lepus Books, London, 1975.

25. Robert Resnick and David Halliday, *Physics Part I,* John Wiley and Sons, Inc., New York, 1966.

26. L.A. Pars, *A Treatise on Analytical Dynamics,* Ox Bow Press, Woodbridge, Connecticut, 1979.

27. William W. Armstrong, ''Recursive Solution to the Equations of Motion of an N-link Manipulator,'' *Proceedings Fifth World Congress on the Theory of Machines and Mechanisms*, pp. 1343-1346, Am. Soc. of Mech. Eng., 1979.

28. J. M. Hollerbach, ''A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation Complexity,'' *IEEE Trans. of Systems, Man, and Cybernetics*, vol. SMC-10, no. 11, pp. 730-736, 1980.

29. Roberto Horowitz, ''Model Reference Adaptive Control of Mechanical Manipulators,'' PhD Thesis, Mechanical Engineering, University of California, Berkeley, California, May, 1983.

30. James U. Korein and Norman I. Badler, ''Techniques for Generating the Goal-Directed Motion of Articulated Structures,'' *IEEE Computer Graphics and Applications*, vol. 2, no. 9, pp. 71-81, November, 1982.

31. J. D. Foley and A. Van Dam, in *Fundamentals of Interactive Computer Graphics*, Addison Wesley, Reading, Mass., 1984.

32. S. Conte and C. de Boor, in *Elementary Numerical Analysis, 3rd edition*, McGraw-Hill Book Company, New York, 1980.

33. Al Pisano, Department of Mechanical Engineering, University of California, Berkeley, CA, 1984. Personal Communication.

34. Jane Wilhelms, ''Virya - A Motion Control Editor for Kinematic and Dynamic Animation,'' *Proceedings of Graphics Interface 86*, pp. 141-146, May, 1986.

35. Brian A. Barsky and Spencer W. Thomas, ''TRANSPLINE -- A System for Representing Curves Using Transformations Among Four Spline Formulations,'' *The Computer Journal*, vol. 24, no. 3, pp. 271-277, August, 1981.

36. Richard H. Bartels, John C. Beatty, and Brian A. Barsky, *An Introduction to the Use of Splines in Computer Graphics,* Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

37. Michael Girard and Antony A. Maciejewski, ''Computational Modeling for the Computer Animation of Legged Figures,'' *SIGGRAPH '85 Conference Proceedings*, vol. 19, pp. 263-270, July, 1985.

38. Lynwood H. Wilson, ''Control Systems Made Simple,'' *Micro/Systems Journal*, pp. 20-24, September/October 1986.

39. Hirofumi Miura and Isao Shimoyama, ''Dynamic Walk of A Biped,'' *International Journal of Robotics Research*, vol. 3, no. 2, pp. 60-74, 1984.

The dynamics of articulated rigid bodies for purposes of animation

William W. Armstrong
and Mark W. Green

Department of Computing Science,
University of Alberta, Edmonton,
Alberta, Canada, T6G 2H1

Curves and surfaces satisfying continuity and smoothness conditions are used in computer graphics to fit spatial data points. In a similar fashion, smooth motions of objects should be available to animators in such a way that the dynamics are correct to the degree required for realism. The motion, like a curve or surface shape, should be controllable by easy manipulations of a set of control parameters or by real-time interaction between the animator and a scene generated by dynamic simulation. In this paper, the objects considered have the form of rigid links joined at hinges to form a tree. This is a reasonable first approximation to human and animal bodies. The equations of motion are formulated with respect to hinge-centered coordinates, and are solved by an efficient technique in time which grows linearly with the number of links.

Key words: Dynamics – Animation – Equations of motion – Robotics

he animation of human and human-like characters is one of the major unsolved problems in computer animation (see for example Badler (1982). The key aspect of this problem is achieving realistic motion with a minimal amount of effort on the part of the animator. Several approaches to human figure animation have been tried in the past with some success.

One of the earliest approaches was to record or digitize the motion of a human (Calvert et al. 1980). The animated human figure mimicked this motion based on the collected data. One of the main problems with this technique was having a properly instrumented human actor perform all the actions required for the animation. Collecting and processing the motion data is a cumbersome task and there is little flexibility in editing the motion once the data has been collected. Another problem with the approach is that the human actor cannot perform dangerous actions (such as falling off a cliff) or simulate the motion of non-human characters. A related approach is to use human body positions as key frames in an animation sequence. This still requires the collection of human movement data (but with a lower volume) and has all the problems associated with key framing (Catmull 1978).

A more recent approach is based on developing a kinematic model of the human body (Zeltzer 1982). This model is based on the anatomy of the human body and characteristics of its motion. Motion is achieved by a hierarchy of motor programs. The low level motor programs control the joint angles for a fixed set of joints. These motor programs are controlled by the middle level motor programs. The middle level motor programs can start and stop the low level programs based on the current state of the model (joint angles, center of mass, support, etc.). Both the low level and middle level motor programs are modeled as finite automata. This approach requires far less effort on the part of the animator but still has some problems. A separate set of motor programs is required for each type of motion. The study of human motion is required to construct these programs. The model can only react to the environment in a restricted way. For example, one of these models could walk over uneven terrain, but could not respond to someone pushing it.

The approach to human figure animation presented here is based on incorporating dynamics into the model of the figure. This added information greatly simplifies interacting with and controlling the model. Once the model has been properly constructed (see Section: Developing figure models) a wide range of motions can be achieved by applying forces or torques to joints in the model at key points in the animation. For repetitive motions these forces can be programmed in the same way as the kinematic models.

In our view of human figure animation a human figure model must have the following characteristics:

1. The model should produce realistic motion sequences when given realistic input data. In other cases the model should produce believable results.
2. The amount of information the animator must provide should be minimal and proportional to the complexity of the motion. If the character is standing still or reacting to the environment in a standard way the animator should not need to specify any motion data.
3. The model should be able to react to and act on its environment. If someone pushes the character it should react to that force. Similarly if the character runs into an object it should respond in a natural way and possibly have some effect on the object (the object moves or falls over). This obviously requires a model that accounts for such physical properties as mass, force, inertia, torque, and acceleration.

In the next three sections of this paper, we develop the dynamics of articulated rigid bodies (skeletons) and an efficient method for solving their equations of motion. In the fifth section, a technique for developing dynamic models of human-like figures is presented. A simple example is used to illustrate this technique. The final section presents some of the ideas we are currently working on.

The equations of motion

In this section, the equations of motion will be presented and explained in a tutorial fashion. The reader is assumed to be somewhat acquainted with the elementary concepts of dynamics: velocity, mass, force, acceleration, angular velocity, moment of inertia, torque, angular acceleration etc. (Goldstein 1959). A treatment of manipulator dynamics based on Lagrangian mechanics is given in a book by Paul 1981. Hollerbach has given a recursive Lagrangian formulation of manipulator dynamics (Hollerbach 1980b) and has discussed Newtonian formulations similar to this one (Hollerbach 1980a). Our treatment is a generalization of work done for the case of linear linkages (Armstrong 1979), with more attention paid to computational efficiency.

Consider a physical quantity, such as a force. It is a vector, and has a meaning independent of any coordinate system. To represent it analytically, one has to introduce a system of coordinates using a frame, which consists of three mutually orthogonal unit vectors. The frames will always be right-handed in this paper. In a given frame, a vector is characterized by three components arranged in a one-column matrix. We shall always use this representation in our formulation of the equations of motion.

For dynamic modeling of linkages, it is convenient to choose frames which move along with the links, as though rigidly attached to them. This is particularly useful since, in the moving frame, certain quantities are constant, such as the representation of the vector from a hinge of the link to the link's centre of mass. To represent positions, we use the vector from the origin of the frame to the position in question. We note, though, that the origin of the frame is unimportant in determining the representation of a vector. In addition to frames which move with the links, we also use a fixed, non-rotating inertial frame.

The transformation of the representation of a vector, a 3-by-1 (column) matrix, from one frame to another is done by multiplying the representation on the left by a 3-by-3 orthogonal matrix. The inverse transformation is done using the transpose of the latter. We shall indicate orthogonal matrices by beginning their names with the letter *R*.

The cross-product of two vectors can be carried out using the representations of the operand vectors and the result in a certain frame, and will be denoted in this paper by the cross symbol ($\times$) as an infix operator.

The unit 3-by-3 matrix will be denoted by **I**. There is a very useful operation on a 3-vector v, with components $v1$, $v2$, $v3$, to get a 3-by-3 matrix V such that for any 3-vector w, the vector $v \times w$ is equal to the product of V and the column-vector w. This is the "tilde" operation:

$$\tilde{v} = V = \begin{bmatrix} 0 & -v3 & v2 \\ v3 & 0 & -v1 \\ -v2 & v1 & 0 \end{bmatrix}.$$

We shall use the following quantities, some of them shown in Fig. 1. Lower-case letters denote scalars (which are subscripted with the appropriate link number) and representations of vectors (super-scripted), while upper-case letters denote matrices. A link, other than the root link of the tree (number 1), has one proximal hinge connecting it to its parent, which is closer to the root, and, except for leaves of the tree, one or more distal hinges. The number of a link is also used to number its proximal hinge. We denote by S_r the set of all links having link r as parent.

Scalar:
m_r the mass of link r;

Representations in the inertial frame:
a_G the acceleration of gravity;
p^r the position vector of the hinge of link r which joins it to its parent, which we shall call the proximal hinge of link r;
v^r the velocity of the proximal hinge of link r;
f_E^r an external force acting on link r at the point p_E^r (see below);
g_E^r an external torque acting on link r;

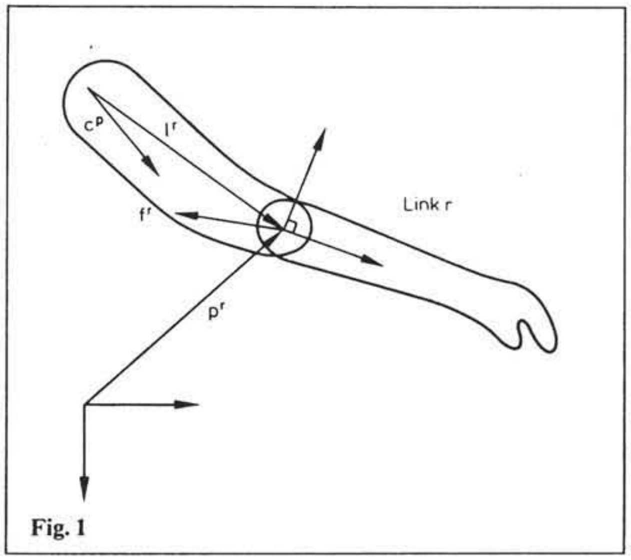


Fig. 1

Representations in the frame of link r:
a^r the acceleration of the proximal hinge of link r;
ω^r the angular velocity of link r;
$\dot{\omega}^r$ its rate of change;
c^r the vector from the proximal hinge to the centre of mass of link r;
f^r the force which link r exerts on its parent at the proximal hinge;
g^r the torque which link r exerts on its parent at the proximal hinge;
p_E^r the vector from the proximal hinge of link r to the point of application of the external force f_E^r to link r;

Representation in the frame of the parent of link r:
l^r the vector from the proximal hinge of the parent of link r to the proximal hinge of link r (constant in the frame of the parent);

Rotation matrices:
R^r converts vector representations in the frame of link r to the representations in the frame of the parent link;
R^{rT} the inverse (= transpose) of R^r;
R_I^r converts from representations in frame r to representations in the inertial frame;
R_I^{rT} the inverse (= transpose) of R_I^r

Matrix representation in frame r:
J^r the moment of inertia matrix of link r about its proximal hinge;

The first equation of motion expresses the fact that the rate of change of angular momentum of link r is equal to the applied torques from various sources.

$$J^r \dot{\omega}^r = g_\Sigma^r - m_r c^r \times a^r + \sum_{s \in S_r} l^s \times R^s f^s \qquad (1)$$

where

$$g_\Sigma^r = -\omega^r \times (J^r \omega^r) - g^r + \sum_{s \in S_r} R^s g^s + R_I^{rT} g_E^r$$
$$+ m_r c^r \times R_I^{rT} a_G + p_E^r \times R_I^{rT} f_E^r. \qquad (2)$$

The right-hand sides of (1) and (2) will now be explained. The part g_Σ^r has been separated out since that will help to explain the solution of the equations as well as accelerate the solution implemented on the computer. The term $-m_r c^r \times a^r$ comes from the fact that the frame in which this equation is formulated is accelerating with respect to the inertial frame, giving the effect of the force $-m_r a^r$ applied at the centre of mass of the link. In the next term, the forces f^s coming from all the sons of link r are first transformed from the coordinates of the son link, where they are represented, to the frame of r by applying R^s. Then the cross product of l^s with the force gives the torque at the proximal hinge of link r due to the force from the son link.

In Eq. (2), the term $-\omega^r \times (J^r \omega^r)$ is a torque coming from the rotation of the frame r with angular velocity ω^r which causes the angular momentum, $J^r \omega^r$, to appear to rotate. This torque term would not appear if the equations had been formulated in the inertial frame. In the inertial frame, however, the inertia matrix J^r would not be constant, making that frame less appropriate for formulating the equations for purposes of computer simulation.

The torque $-g^r$ is the negative of the torque which, by definition, link r exerts on its parent at its proximal hinge. It is the parent's reaction which is "equal and opposite". To be added in next are the torque terms coming from the son links, which must be converted from their representations in the sons' frames, and the external torque, which must be converted from its representation in the inertial frame. It doesn't matter where the torques are applied, since the links are considered rigid. Finally, the force of gravity, $m_r a_G$, converted from the inertial frame, causes a torque at the proximal hinge of link r when applied at its centre of mass; and similarly for the external force f_E^r acting at p_E^r.

The next equation of motion gives the force f^r acting on the parent of link r at the proximal hinge of link r. As above, it is convenient to separate out a term f_Σ^r which does not involve the quantities a^r or others which depend on it according to our solution method.

$$f^r = f_\Sigma^r - m_r a^r + m_r c^r \times \dot{\omega}^r + \sum_{s \in S_r} R^s f^s \tag{3}$$

where

$$f_\Sigma^r = -m_r \omega^r \times (\omega^r \times c^r) + R_I^{r\,T} (f_E^r + m_r a_G). \tag{4}$$

In Eq. (3), the term $-m_r a^r$ comes from the fact that the frame r is accelerating, and the next term from the fact that it is rotating at an accelerating angular velocity, which causes the centre of mass to accelerate with respect to the inertial frame. Finally, the forces from the son links are converted to frame r and communicated to the parent at the hinge.

In Eq. (4), we get first a centrifugal force from the rotation of the frame and second a contribution from the external force and gravity acting on the link.

The last equations describing the motion relate the acceleration at the proximal hinge of a son link s of link r to the linear and angular accelerations at the proximal hinge of r:

$$R^s a^s = a_c^s + a^r - l^s \times \dot{\omega}^r \tag{5}$$

where

$$a_c^s = \omega^r \times (\omega^r \times l^s). \tag{6}$$

In summary, then, the equations of motion for each link are formulated in terms of a moving frame attached to the proximal hinge of the link, and consist of equations giving the effect of torques (1), (2), equations giving the effect of forces (3), (4), and equations relating the accelerations at parent and son nodes (5), (6). In order to get the motion, we must solve these equations either to get the torques given the motion, which would be reasonable to control the linkages along a prescribed path, or to get the motion (accelerations, velocities, positions and orientations of the links through time) given the torques at the hinges and the external forces and torques. We choose the latter approach, since our aim is ultimately to let the animator control the motion by application of torques using hand and other controllers in real time. The previous approach has been treated by Luh et al. 1979.

Solution of the equations of motion

Methods for converting the equations mentioned at the beginning of this paper to a matrix form are well-known. The resulting square matrix can be quite large, with a line for every degree of freedom in the system. Here we have six degrees of freedom for link 1, the root, three for position and three for orientation, and three degrees of freedom for the orientation of every other link (the hinges constrain the position, but not the orientation in our case). A simple model of the human body without hands or feet has at least 12 links, leading to a 39-by-39 matrix. We can expect this number to increase drastically if the links are required to be flexible or deformable in any way. The growth in terms of the number of links is quadratic by that technique. The method we are going to use grows linearly with the number of links, and is appropriate, we feel, for animation purposes, where the number of links may be large.

The hypothesis which aids in solving the equations is that there are linear relationships between the acceleration a^r of the link and the amount of angular acceleration it undergoes and between a^r and the reactive force on the parent. Think of giving the configuration of links distal to a certain hinge a push at the hinge which causes a certain acceleration. This will cause a certain angular acceleration and a certain reactive force, which can be expressed as follows:

$$\dot{\omega}^r = K^r a^r + d^r \tag{7}$$

$$f^r = M^r a^r + f'^r. \tag{8}$$

If we assume this linearity for all the sons of link r, then we can show it inductively for link r, and at the same time develop a computational method for solving the equations of motion by calculating the "recursive" coefficients K^r, d^r, M^r, and f'^r from the distal links towards the root. We encourage the reader to write down Eq. (1) and carry out the following simple substitutions: First the quantity f^s of a son link s is replaced by the corresponding expression from (8) with s instead of r. Then the acceleration a^s appearing therein is rewritten using the right hand side of (5) prefixed by R^{sT}. The cross product $l^s \times \dot{\omega}^r$ is replaced by the product of the matrix $\tilde{l}^s$ with the column vector $\dot{\omega}^r$. This enables us to collect the terms in $\dot{\omega}^r$ and obtain

$$\dot{\omega}^r = T^r(g^r_{\Sigma} - m_r c^r \times a^r$$
$$+ \sum_{s \in S_r} \tilde{l}^s R^s (f'^s + M^s R^{sT}(a^s_c + a^r))) \tag{9}$$

where we have set

$$T^r = (J^r + \sum_{s \in S_r} \tilde{l}^s R^s M^s R^{sT} \tilde{l}^s)^{-1}. \tag{10}$$

Now we can determine the recursive coefficients for (7):

$$K^r = T^r(\sum_{s \in S_r} \tilde{l}^s R^s M^s R^{sT} - m_r \tilde{c}^r) \tag{11}$$

and

$$d^r = T^r(g^r_{\Sigma} + \sum_{s \in S_r} \tilde{l}^s R^s (f'^s + M^s R^{sT} a^s_c)). \tag{12}$$

The next step is to determine the recursive coefficients for the force f^r. We first substitute in (3) for the forces f^s using (7) for s instead of r. The quantity a^s is replaced using (5), and the resulting $\dot{\omega}^r$ values are replaced using (7).

This gives

$$M^r = -m_r \mathbf{I} + m_r \tilde{c}^r K^r$$
$$+ \sum_{s \in S_r} R^s M^s R^{sT}(\mathbf{I} - \tilde{l}^s K^r) \tag{13}$$

and

$$f'^r = f^r_{\Sigma} + m_r c^r \times d^r$$
$$+ \sum_{s \in S_r} (R^s f'^s + R^s M^s R^{sT}(a^s_c - l^s \times d^r)). \tag{14}$$

Computation of the solution to the equations

There are certain frequently used constants which should be precomputed:

$$J^r, m_r \mathbf{I}, m_r c^r, m_r \tilde{c}^r, m_r a_G, \tilde{l}^r.$$

At each iteration step, the equations of motion must be solved for the angular accelerations of the links by first determining the coefficients of the linear expressions (7) (8) in a pass inward from the distal links to the root link. Then the Eq. (8) for the root link, number 1, can be used to obtain the acceleration a^1, since f^1, the force on the parent of the root link, is zero (there is no parent). Eq. (5) can then be used repeatedly to get the acceleration of the son links, whose angular accelerations are computed using (7), now with known coefficients, on a pass outwards from the root link. The constraint forces f^r at the hinges can also be computed during this pass, but this is not required unless we are checking the solution.

After the solution phase at each time step, comes the integration phase, where $\dot{\omega}^r$ is multiplied by the time step δt to get the increment of ω^r, and the latter is multiplied by δt to get an incremental rotation vector for the link. The incremental rotation vectors are used to update

the rotation matrices which specify the orientations of the links: R_I^r, R^s (for son links only). The linear acceleration and velocity of link 1 can also be determined as well as its position, which, together with the orientations of all links, given by R_I^r, allows all the hinge positions in the inertial frame to be determined (for the purpose of graphics display, for example).
There is one deficiency in the above method, however, namely that some of the quantities which are computed at each integration step can be expected to vary slowly, and hence recomputation is not required at every increment δt. Hence all the computations will now be divided into two bands: a "fastband" executed at each time step, and a "slowband" executed completely once every v executions of the fastband. The slowband computation can be split into v parts and executed uniformly over the fastband iterations in a real-time system.
The resulting organization of the computation is as follows:
Slowband computations inbound (i.e. from the leaves to the root):
Do for each link r starting at the leaves of the tree:

$$\textit{Compute}\ a_c^s=\omega^r\times(\omega^r\times l^s) \tag{15}$$

$$\textit{Compute}\ Q^s=R^s M^s R^{sT}\ \text{for}\ s\in S_r \tag{16}$$

$$\textit{Compute}\ W^s=\tilde{l}^s Q^s\ \text{for}\ s\in S_r \tag{17}$$

$$\textit{Compute}\ T^r=(J^r+\sum_{s\in S_r} W^s\tilde{l}^s)^{-1} \tag{18}$$

$$\textit{Compute}\ K^r=T^r(\sum_{s\in S_r} W^s-m_r\tilde{c}^r) \tag{19}$$

$$\textit{Compute}\ M^r=(m_r\tilde{c}^r)K^r-m_r\mathbf{I}$$
$$+\sum_{s\in S_r} Q^s(\mathbf{I}-\tilde{l}^s K^r) \tag{20}$$

$$\textit{Compute}\ g_\Sigma^{1r}=-\omega^r\times(J^r\omega^r)+R_I^{rT}g_E^r$$
$$+p_E^r\times R_I^{rT}f_E^r+(m_r c^r)\times R_I^{rT}a_G \tag{21}$$

(assuming f_E^r and g_E^r are slowly-varying).

$$\textit{Compute}\ f_\Sigma^r=-\omega^r\times(\omega^r\times(m_r c^r))$$
$$+R_I^{rT}(f_E^r+m_r a_G) \tag{22}$$

(assuming f_E^r is slowly-varying).

Fastband computations, inbound:
Note: at this point the hinge torque values g^r from the controls manipulated by the animator are inserted. One approach to controlling these values is presented in the following sections, and others are under development.
Do for each link r, starting distally and proceeding towards the root:

$$\textit{Compute}\ g_\Sigma^r=g_\Sigma^{1r}-g^r+\sum_{s\in S_r} R^s g^s \tag{23}$$

$$\textit{Compute}\ d^r=T^r(g_\Sigma^r+\sum_{s\in S_r} l^s\times(f''^s+Q^s a_c^s)) \tag{24}$$

$$\textit{Compute}\ f'^r=f_\Sigma^r+(m_r c^r)\times d^r$$
$$+\sum_{s\in S_r}(f''^s+Q^s(a_c^s-l^s\times d^r)) \tag{25}$$

$$\textit{Compute}\ f''^r=R^r f'^r. \tag{26}$$

Fastband computations, outbound:

$$\textit{Compute}\ a^1=-(M^1)^{-1}f'^1 \tag{27}$$

$$\textit{Compute}\ \dot{\omega}^1=K^1 a^1+d^1. \tag{28}$$

Do the following for each link r other than the root outbound:

$$\textit{Compute}\ a^r=R^{rT}(a_c^r+a^p-l^r\times\dot{\omega}^p) \tag{29}$$

where p is the parent of the link r.

$$\textit{Compute}\ \dot{\omega}^r=K^r a^r+d^r \tag{30}$$

$$\textit{Compute}\ f^r=M^r a^r+f'^r \tag{31}$$

if required for a check of the solution.
Integration, fastband:

$$\textit{Compute}\ \omega^r+\delta t\,\dot{\omega}^r \tag{32}$$

and assign to ω^r.

$$\textit{Compute}\ \delta u^r+\delta t\,\omega^r \tag{33}$$

and assign to δu^r, where the latter quantity accumulates a small rotation of link r as an "infinitesimal vector".

$$\textit{Compute}\ v^r+\delta t\,R_I^r a^r \tag{34}$$

and assign to v^r.

$$\textit{Compute}\ p^r+\delta t\,v^r \tag{35}$$

and assign to p^r.
Actually, Eqs. (34) and (35) are used only for the root link. The positions of the other links are determined from the position of the root hinge and from the hinge-to-hinge vectors in the inertial frame $R_I^r l^s$, where r is the parent of

s. This prevents accumulation of errors so that the graphic display will not become distorted.

Integration, slowband:
Here the rotation matrices are updated, a costly process which should be done only when the errors introduced by tardy updating are sufficiently small.

Do the following for each link starting distally:

$$\textit{Compute}\ R_l^r(\mathbf{I}+\delta\tilde{u}^r) \tag{36}$$

and assign to R_l^r. When this has been done δu^r must be reset to the zero vector to begin its accumulation again during the fastband integration. Also compute and store the transpose $R_l^{r\,T}$.

$$\textit{Orthonormalize}\ R_l^r \tag{37}$$

to prevent accumulation of errors which would ultimately destroy the simulation.

$$\textit{Compute}\ R_l^{r\,T} R_l^s \tag{38}$$

for every $s \in S_r$, and assign to R^s. Also compute and store the transpose $R^{s\,T}$.
In implementing the above solution, it is of critical importance to choose an integration step size δt several times smaller than the period of any frequency present in the system. Otherwise numerical instability will destroy the simulation. (It is amusing to watch the stick figure's arms begin to oscillate wildly as it flies off to oblivion though!) The highest frequencies will probably be the rotations of small links about their long axes under the influence of elastic torques at the hinges which maintain the alignment of the links. What one can do to keep these frequencies low is to falsely enlarge the moments of inertia about these axes. It is an open question to what extent this can be done without destroying the realism of the simulation. Such tricks are often applied in aircraft simulators, for example, and animation can undoubtedly benefit from using them heavily. A method for removing two degrees of freedom from the hinges has been described elsewhere (Armstrong 1979).
Other factors which must be appropriately adjusted are the frictions of the hinges, which create viscous damping torques proportional to the relative angular velocities of the two links involved. As we shall see, creating a zone of free play at the hinges, where the stiffness of the hinge has no effect, allows the friction to absorb the energy erroneously introduced by numerical inaccuracies, leading to a more stable model.

Developing figure models

This section describes one approach to developing human figure models based on the dynamics presented in the previous three sections. These models represent the positions and orientations of the figure along with how it behaves when it interacts with its environment. This approach is illustrated by two examples presented at the end of this section.
Our approach to human figure modeling consists of three steps. The first step is developing a skeleton for the figure. This skeleton consists of a set of links representing the figure's limbs and a set of joints representing the places where the limbs are connected. The set of links representing the figure form a tree with each link having at most one parent. The skeleton represents the topology and physical properties of the figure. The skeleton itself can be viewed as a stick figure. The graphical display of the figure used in an animated sequence is generated from the skeleton. Techniques for converting skeletons to graphical displays can be found in Burtnyk and Wein 1976; Zeltzer 1982 and Lake 1985. Associated with each link in the model is its mass, center of mass, and inertia. These quantities must be calculated or estimated from the figure being modeled.
In order to have realistic motion the skeleton must be augmented with additional information pertaining to the behavior of its joints. The second step in this modeling process is determining this information and incorporating it into the figure model. This step is divided into two parts. In the first part the support for the skeleton is developed. The force of gravity is continuously acting on the figure and unless it is counteracted the figure will fall or collapse. There are several ways in which the figure can be supported. One way is to attach an upward pointing force (on average equal to the mass of the object times the acceleration of gravity) to one of the links in the model. The link used depends upon the nature of the animation. For walking and similar actions the force is at-

tached to the foot that is in contact with the ground. For gymnastics the force might be attached to the center of mass of the figure. Another approach to counteracting the force of gravity is discussed in the first example at the end of this section.

The joints in the model cannot be free to move any way they wish. Each joint has a certain range of angles it is capable of moving through. Angles outside of this range will appear unnatural. One way of constraining the motion of the joints is to stop the joint rotation when it reaches the end of its natural range. This is not a satisfactory solution for the following reasons. First, it leads to very unnatural motions. When people move their limbs there is a definite acceleration and deceleration at the end of the motion, they don't come to an abrupt stop at the end of the motion (Schmidt 1982). Second, the abrupt changes caused by the clamping of the rotation angles violates the assumptions used in the development of the equations presented in Section: Solution of the equations of motion. Third, sometimes the animator wants a limb to move outside its normal range. If a large enough force is applied to the model, one or more of the limbs should move out of its natural range.

The approach to joint behavior that we have developed is based on determining the natural range of the joint and then applying torques about the joint to keep it in this range. At each time step the torque applied to a limb is a function of the angle between it and its parent. This torque will move the limb towards the center of its normal range. The torque functions we use in our models have the shape shown in Fig. 2. As can be seen from this figure there is a range of angles where no restorative torques are applied. This area of free play leads to more natural motions and decreases the tendency of the limbs to oscillate. The size of the free play zone and the slope of the function at the ends of this zone determine the behavior of the joint. A narrow free play zone with steep sides will give rise to a stiff joint. The free play zone can be moved around in order to simulate different types of motion. Note that there are torques that will cause the joint to act unnaturally.

At this point if the model is placed in a simple environment with only gravity it will eventually reach equilibrium (in practice this occurs after one or two seconds of simulation time depending on how close the initial state of the model is to equilibrium). If the environment acts on the model it will essentially roll with the punches. The third step in our modeling approach is determining the external forces and torques required for specific motions such as walking, throwing, or diving. In most cases these forces and torques are spikes applied at fixed intervals of time or when certain conditions hold.

As an example of how this approach works consider the problem of animating a finger tapping on a table. The skeleton for this finger is shown in Fig. 3. This skeleton has five links, three of which are visible in the figure. The middle three links represent the visible part of the finger and for the purposes of this example all have the same length and mass. At the left end of the figure is a link of length zero representing the attachment of the finger to the hand. This end of the finger should have a relatively constant position, therefore, it has a much higher mass and inertia (about 100 times) than the other links. In order to support the finger a variable force is applied to this link. The strength of this force is inversely proportional to the distance between the current po-

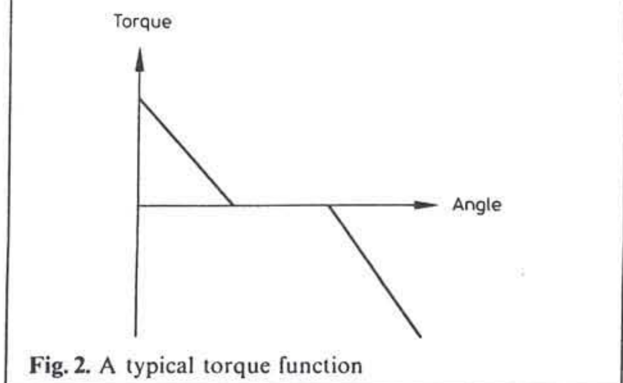


Fig. 2. A typical torque function

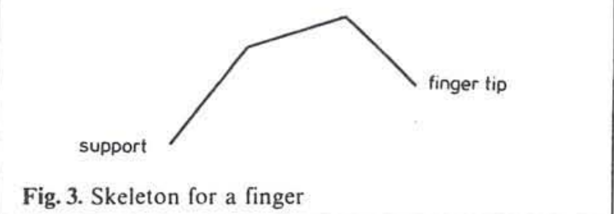


Fig. 3. Skeleton for a finger

sition of the link and its desired position. The fifth link is at the other end of the finger and also has zero length. This link serves as a convenient place for applying an external force.

Next the torque functions for the joints must be determined. In this case we ran the simulation for several seconds without torque functions and recorded the joint angles that appeared natural. The torque functions were based on these observations and several runs to determine appropriate slopes of the functions.

Finally, in order for the finger to tap, an external force must be periodically applied to it. In this case, a spike lasting 0.11 s is applied to the finger tip every 2 s. The model is allowed 2 s to stabilize before the first spike is applied.

The simplicity of this example indicates the ease with which realistic animation can be produced with this approach.

This example has been run on both a SUN workstation (a MC68010 with no floating point hardware) and a VAX 11/780 with a floating point accelerator. A time step of 0.01 s was used in these runs and the slow band was calculated at every time step. On the SUN, 604 cpu seconds were required to produce an 8-s animated sequence of the finger tapping (a wire frame perspective display of the skeleton was produced every 0.1 s). On the SUN hardware the simulation is a factor of 75 slower than real time. The same program on the VAX requires 89 s to produce 8 s of animation. If the slow band is calculated every other time step, the SUN requires 385 s and the VAX requires 56.6 s to produce 8 s of animation. On the VAX this is close enough to real time to get some feel for the motion while it is being calculated.

The second example is based on a human body model consisting of twelve links. Figure 4 shows a sequence of images simulating the first 0.5 s of a diving motion very simply defined by applying external forces downward and forward on the head and arms, forward on the lower body and upward and backward on the feet. The hinge torques generated by stiffness and friction were sufficient to maintain reasonable behavior without any intervention by joint motor programs. The moments of inertia of the links were increased by about an order of magnitude from the correct values for the large links, and by three orders of magnitude for the small links (like the neck). Further experiments are required to determine the allowable falsification of the moments of inertia still giving reasonable behavior. With this falsification of the true dynamics, the integration step size could be taken to be 1/30 s. It was possible to generate one second of motion on the VAX 11/780 in 9.5 s of CPU time, even if the slowband computations were done at every step. This was about 10 times slower than real time. Doing the slowband computations every five fastband cycles reduced this to five times real time.

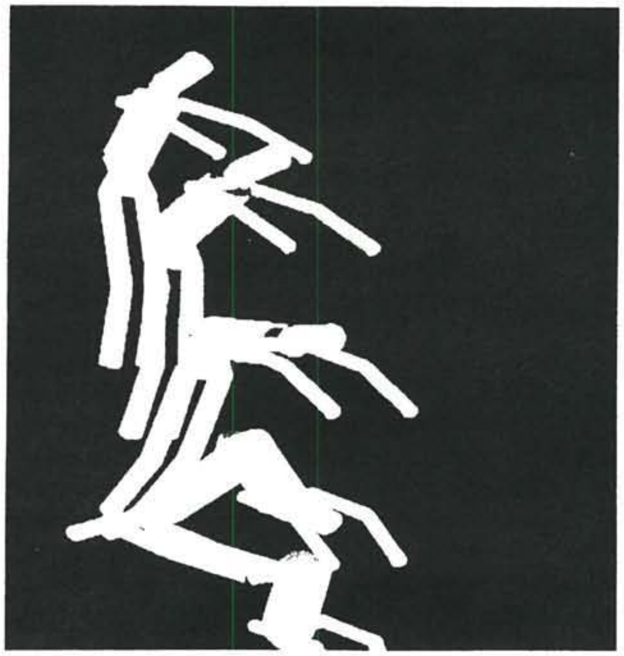


Fig. 4

The public domain software "DynaTree", written in C by W.W. Armstrong, carried out the dynamics computations. The shaded color display of the resulting file of positions, requiring about four minutes per position, was done using software developed by Robert Lake.

Clearly, we have a long way to go before having near-real-time simulation and display of shaded images, but stick-figure simulation in real time is very close to being a reality. Dropping some of the non-essential terms in the equations of motion would probably be enough to attain it. We are thus already facing the question of how the animator can control the motion in near-real-time.

Conclusions and further work

In this paper we have presented the dynamics of articulated rigid bodies, an efficient method for solving their equations of motion, and a technique for developing human figure models based on these dynamics. The examples in Section: "Developing figure models" show that these equations can be solved almost in real time.

This work suggests a number of topics for further research. One possible topic is reducing the time required to solve the equations of motion. Two possible approaches to this problem are developing better implementations and designing special hardware for their execution. Another research topic is developing better techniques for controlling the figure. At the present time we have not thoroughly explored all the possibilities for supporting the model against the force of gravity. Also means for controlling complicated motions need to be developed. Finally the figure models need to be integrated with an environment where they can interact with each other and long range planning can occur.

References

Armstrong WW (1979) Recursive Solution to the Equations of Motion of an N-link Manipulator. Proc Fifth World Congress on Theory of Machines and Mechanisms, Montreal, vol 2. Am Soc Mech Eng 1343-1346

Badler N (1982) (ed) Special Issue of IEEE Computer Graphics and Applications on Human Body Models and Animation, vol 2, no 9

Burtnyk N, Wein M (1976) Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key Frame Animation. CACM 19:564

Calvert TW, Chapman J, Patla A (1980) The Integration of Subjective and Objective Data in the Animation of Human Movement. Siggraph '80, Proceedings, p 198

Catmull E (1978) The Problems of Computer-Assisted Animation. Siggraph '78 Proceedings, p 348

Goldstein H (1959) Classical Mechanics. Addison-Wesley, Reading MA.

Hollerbach JM (1980a) An Iterative Lagrangian Formulation of Manipulator Dynamics. Massachusetts Institute of Technology, AI Laboratory, A.I. Memo No 533. Revised March 1980

Hollerbach JM (1980b) A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation. IEEE Trans. on Systems, Man and Cybernetics, SMC-10, 11, pp 730-736

Lake R (1985) Producing a Shaded Tubular Man. Department of Computing Science, University of Alberta

Luh J, Walker M, Paul R (1979) On-line Computational Scheme for Mechanical Manipulators. 2nd IFAC IFIP Symposium on Information Control Problems in Manufacturing Technology, Stuttgart, Germany

Paul P (1981) Robot Manipulators: Mathematics, Programming, and Control. MIT Press, Cambridge, MA

Schmidt RA (1982) Motor Control and Learning: A Behavioral Emphasis. Human Kinetics Publishers, Champaign Illinois

Zeltzer D (1982) Motor Control Techniques for Figure Animation. IEEE Computer Graphics and Applications 2:53

Computational Modeling for the Computer Animation of Legged Figures

Michael Girard
and
A. A. Maciejewski

Computer Graphics Research Group
The Ohio State University

OSU CGRG /Cranston Center
1501 Neil Avenue
Columbus OH 43201

Abstract

Modeling techniques for animating legged figures are described which are used in the PODA animation system. PODA utilizes pseudoinverse control in order to solve the problems associated with manipulating kinematically redundant limbs. PODA builds on this capability to synthesize a kinematic model of legged locomotion which allows animators to control the complex relationships between the motion of the body of a figure and the coordination of its legs. Finally, PODA provides for the integration of a simple model of legged locomotion dynamics which insures that the accelerations of a figure's body are synchronized with the timing of the forces applied by its legs.

CR Categories and Subject Descriptors: 1.3.7 [Computer Graphics]: Graphics and Realism: Animation. Additional Key Words and Phrases: motion control, computational modeling, manipulators, legged locomotion

Introduction

The problems of animating articulated figures with multiple legs have long been a source of difficulty in the computer animation field. Joint angle interpolation between "key" joint positions is the most widely used method of animating jointed animals. This method fails to work, however, for cases in which the end of a limb must be constrained to move along a particular path - the interpolated joint positions of two "key" leg positions planted on the ground will not, in general, remain on the ground (fig. 1).

Another difficulty is the sheer tedium of positioning "keys" for limbs containing many degrees of freedom. The animal shown in figure 2 possesses 9 degrees of freedom in each leg,9 degees of freedom in the neck, and 18 degees of freedom in the "spine." An animator using a key joint system would have to manage positioning a total of 63 joints.

A further problem is that a walking or running figure is more than an assemblage of moving limbs - the coordination of legs, body and feet are functionally related in a complex fashion. The motion of the body of a figure and the timing and placement of legs are both kinematically and dynamically coupled.[20-36]

The approach taken in the design of the PODA system is to provide the animator with a computational model which facilitates the integration and direct control of the functional dependencies between different parts of a figure. An interactive menu-driven interface is used for both the incremental construction and behavioral control of animals possessing any number of legs composed of any number of joints. A strategy is implemented in which the figure's motion may be designed and manipulated at different levels of control. At the lowest level the animator may define and adjust the character of the movement of the legs and feet. At a higher level the animator may direct the coordination of the legs and control the overall motion dynamics and path of the body.

The primary goal of our efforts is to build a framework in which the synthesis of legged figure motion may be artistically conceived and controlled at increasingly higher levels of complexity and abstraction. In this regard, our initial efforts have been focussed on developing a general model for legged locomotion due to its importance to the execution of more complex motor skills, such as those which are required for dance and gymnastics[27].

In this first section we will outline the solution taken in PODA for the control of single limbs. This will set the stage for the discussion of the legged locomotion model which utilizes the limb control methods described.

The research described in this paper was supported, in part, by National Science Foundation grant DCR-8304185, and in part, by a National Science Foundation Fellowship.

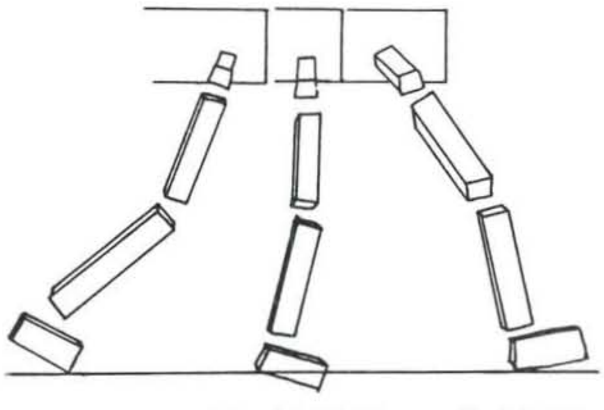


Figure 1

THE CONTROL OF LIMBS

Representing articulated limbs

In order to define the functionality of an arbitrary articulated figure, PODA has adopted the kinematic notation presented by Denavit and Hartenberg [3]. This specifies a unique coordinate system for every individual degree of freedom present in the figure. These degrees of freedom, whether rotary or prismatic, will be referred to as joints and the fixed interconnecting bodies as links. The four parameters used to define the transformation between adjacent coordinate systems are the length of the link a, the twist of the link α, the distance between links d, and the angle between links θ. The single variable associated with the transformation depends on the type of joint represented, that is θ for rotary or d for prismatic joints (fig. 3).

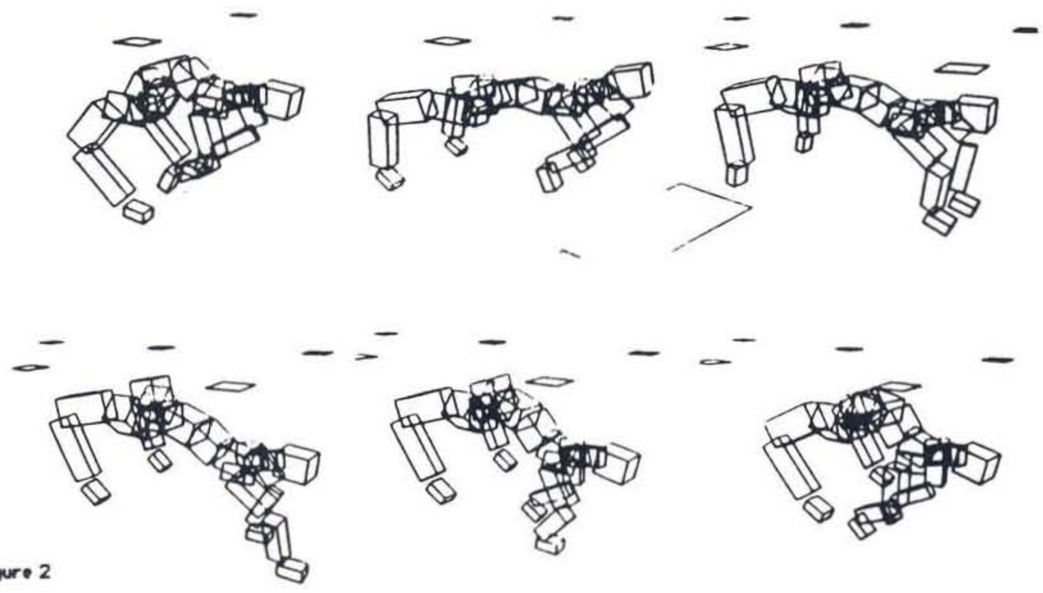


Figure 2

Given the above definitions, it can be shown [12] that the transformation between adjacent coordinate frames $i-1$ and i denoted by ${}^{i-1}T_i$ is given by the homogeneous transformation:

$$ {}^{i-1}T_i = \begin{bmatrix} \vec{n} & \vec{o} & \vec{a} & \vec{p} \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

where

$$ \begin{aligned} p_x &= a_i \cos\theta_i & n_x &= \cos\theta_i \\ p_y &= a_i \sin\theta_i & n_y &= \sin\theta_i \\ p_z &= d_i & n_z &= 0 \\ o_x &= -\cos\alpha_i \sin\theta_i & a_x &= \sin\alpha_i \sin\theta_i \\ o_y &= \cos\alpha_i \cos\theta_i & a_y &= -\sin\alpha_i \cos\theta_i \\ o_z &= \sin\alpha_i & a_z &= \cos\alpha_i \end{aligned} $$

By repeatedly applying adjacent link transformations the relationship between any two coordinate systems i and j is easily obtained using

$$ {}^{i}T_j = {}^{i}T_{i+1}\,{}^{i+1}T_{i+2}\cdots{}^{j-1}T_j $$

The above equation permits any link of the figure, defined in its own coordinate frame, to be represented in any arbitrary reference or world coordinate frame. More importantly, it can be used to determine the number and characteristics of the degrees of freedom available for simulating coordinated movement.

The Jacobian Matrix

Given the above framework, one can easily see that, given the state of the joint angle variables, we can compute the position of all of the links and arrive at the position of the end of the limb. This is called the *forward kinematics* problem. The reverse situation, that of computing the joint angles from the position of the end of the limb, is necessary if we wish to place a foot or hand in some desired place–what Korein and Badler have called "goal directed motion" [32]. This is called the *inverse kinematics* problem. The legged locomotion models in PODA rely heavily on the need for goal-directed motion: feet must be moved along trajectories, placed exactly at desired footholds and held in place as the body passes over them.

The solution of the inverse kinematics problem is the source of much of the difficulty in dealing with controlling articulated figures. A general solution for arbitrary articulated chains does not exist and even those that lend themselves to an analytic solution result in nonlinear equations [12]. Additional complications are incurred when redundant degrees of freedom are present.

The solution adopted by PODA is to linearize the equations about the current operating point. The six-dimensional vector representing an incremental change in position and orientation in three space of an arbitrary link is linearly related to the vector $\Delta\vec{\theta}$ by the Jacobian matrix J through the equation

$$ \Delta\vec{x} = J(\vec{\theta})\Delta\vec{\theta} \tag{1} $$

for changes which are sufficiently small. Thus by updating the Jacobian each cycle time, the advantages of a linear system are obtained. This allows the application of all of the techniques of solving linear equations to obtain the desired result (to be discussed in the next section). The use of Jacobians has long been a common practice in nonlinear control system theory and has been successfully applied in the field of robotics [18,19].

Due to its central nature in the animation of articulated figures, an efficient implementation for generating the Jacobian is essential to a viable system. While there are may different techniques available, a particularly elegant method has been formulated with the use of screw motor variables [17]. A screw motor is characterized by the variables $\vec{\omega}$, the angular velocity of the screw axis, and $\vec{\mu}$, the velocity of a point attached to the screw axis which coincides with the origin of the world coordinate frame. In terms of these variables, the desired displacement of the foot may be expressed as

$$ \begin{aligned} \vec{\omega} &= R\vec{\omega}_f \\ \vec{\mu} &= R\vec{v}_f - \vec{\omega}\times\vec{p} \end{aligned} \tag{2} $$

with the original foot displacement $\Delta\vec{x}$ given by

$$ \Delta\vec{x} = \begin{bmatrix} \vec{v}_f \\ \vec{\omega}_f \end{bmatrix} . $$

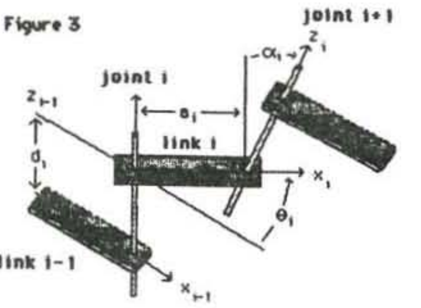
where R is the upper 3×3 rotation partition of the homogeneous transformation describing the desired point whose velocity is being specified and $\vec{p}$ is the position of this point given by the fourth column of its homogeneous transformation. It can be shown [16] that the Jacobian is given by

$$ J = \begin{bmatrix} p_1\times a_1 & p_2\times a_2 & \dots & p_n\times a_n \\ a_1 & a_2 & \dots & a_n \end{bmatrix} \tag{3} $$

where a_i and p_i are the third and fourth columns, respectively, of the homogeneous transformation matrix ${}^{0}T_{i-1}$. The first column of the Jacobian is given by

$$ p_1 = [0\,0\,0]^{\mathsf{T}} \qquad a_1 = [0\,0\,1]^{\mathsf{T}}. $$

Figure 3



Link Parameters Associated With Link i

This formulation of the Jacobian allows a minimal amount of extra computation since the majority of the work has already been done in generating the homogeneous transformations required to display the object. This is in contrast with other techniques which do not express the Jacobian in the world coordinate frame [37].

Inverting the Jacobian

Given a linear system by virtue of the Jacobian, we now need to invert the relationship represented by equation (1) in order to determine the required $\Delta\vec{\theta}$ to achieve a desired $\Delta\vec{x}$. Since we are dealing with arbitrary articulated figures, the Jacobian is in general not square and therefore its inverse is not defined. To obtain a useful solution regardless of the rank of J, the pseudoinverse is applied. The pseudoinverse will be denoted by J^+ and is the unique [13] matrix which satisfies the four properties:

$$\begin{aligned} J\,J^+\,J &= J \\ J^+\,J\,J^+ &= J^+ \\ (J^+\,J)^\top &= J^+\,J \\ (J\,J^+)^\top &= J\,J^+ \end{aligned} \tag{4}$$

The advantages of using the pseudoinverse lie in that it returns the least squares minimum norm solution to equation (1). Thus it provides useful results in both the under and over determined cases. Other generalized inverses may also be applied [1,2]. An excellent overview concerning the pseudoinverse control of redundant manipulators as well as a geometric interpretation of the pseudoinverse using singular value decomposition is presented in [7].

A number of different methods for calculating pseudoinverses have been discussed in the literature [6,14]. A discussion of the some of the numerical considerations involved in computing the pseudoinverse is presented by Noble [11]. The simplest expressions for a pseudoinverse appear for matrices known to be of full rank. For an $m \times n$ matrix A of rank r, the expression for the pseudoinverse is given by

$$A^+ = \begin{cases} (A^\top A)^{-1} A^\top & \text{if } m > n = r \text{ and} \\ A^\top (A\,A^\top)^{-1} & \text{if } r = m < n \end{cases} \tag{5}.$$

The use of Gaussian elimination with pivoting removes the need for an explicit inverse calculation and results in a stable and efficient technique for computing the pseudoinverses under these conditions.

For matrices of unknown rank a recursive procedure for computing the pseudoinverse presented by Greville [4] may be used, the details of which are beyond the scope of this paper.

Controlling Redundant Limbs in PODA

While the above section illustrates how the pseudoinverse can be used to obtain a useful solution to equation (1), for cases where redundant degrees of freedom exist, it is only one of an infinite number of solutions. The manner in which the animator is given the flexibility to determine which of the available solutions is most desirable is through a projection operator. It can be shown that shown [5] that the general solution of equation (1) is given by

$$\Delta\vec{\theta} = J^+\Delta\vec{x} + (I - J^+\,J)\vec{z} \tag{6}$$

where I is an $n \times n$ unity matrix and z is an arbitrary vector in $\Delta\theta$-space. Thus the homogeneous portion of this solution is described by a projection operator $(I - J^+\,J)$ that maps the arbitrary vector $\vec{z}$ into the null space of the transformation. The physical interpretation of the homogeneous solution is illustrated in figure 4.

Thus by different choices for the vector $\vec{z}$, various desirable properties described in θ-space can be achieved under the constraint imposed by exact achievement of the specified $\Delta\vec{x}$. One particularly useful property is to keep joints as close as possible to some particular angles chosen by the animator. This is done [8] by specifying the vector $\vec{z}$ in equation (6) to be

$$\vec{z} = \nabla H$$

with

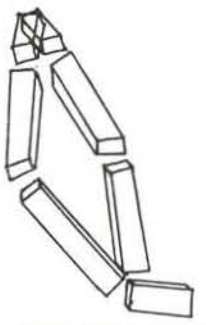
$$H = \sum_{i=1}^{n} \alpha_i(\theta_i - \theta_{c_i})^2$$

where θ_i is the ith joint angle, θ_{c_i} is the center angle of the ith joint angle, and α_i is a center angle gain value between zero and one. The equation may also be generalized for H equal to any smooth function one wishes to minimize.

The center angles define the desired joint angle positions and their associated gains define their relative importance of satisfaction. From the animators point of view, the gains may be thought of as "springs" which define the stiffness of the joint about some desired center position. PODA provides interactive specification of center angles and gains as a means of controlling redundant degrees of freedom in the legs.

The implementation of this formulation can be included in the gaussian elimination procedure for computing the pseudoinverse if it is properly decomposed [7].

Figure 4



Homogeneous solution to the Jacobian equation is the set of joint velocities which cause no end effector motion.

MODELING THE KINEMATICS OF LEGGED LOCOMOTION

The task of a kinematic model for legged locomotion is to coordinate the motion of the legs,feet and body in terms of their respective positions and velocities (Newtonian mechanical properties such as force and mass are not considered). The kinematic model must enable the animator to design the timing relationships between the legs and the character of the steps taken by each leg in accordance with the design of the body's trajectory, orientation and speed. Ideally, it should be easily adaptable to any extensions made in the dynamics domain.

Gait Design in PODA

The model of locomotion implemented in PODA utilizes a number of parameters which are convenient for describing the gait of a figure– the terms and relations are derived from robotics research on walking machines [22-26].

A *gait pattern* describes the sequence of lifting and placing of the feet. The pattern repeats itself as the figure moves: each repetition of the sequence is called the *gait cycle*.

The time (or number of frames) taken to complete a single gait cycle is the *period* P of the cycle.

The *relative phase of leg i*, R_i , describes the fraction of the gait cycle period which transpires before leg i is lifted. The relative phases of the legs may be used to classify the well known gaits of quadrupedal animals (fig. 5).

During each gait cycle period any given leg will spend a percentage of that time on the ground–this fraction is called the *duty factor of leg i*. For example, the duty factor may be used to distinguish between the walking and running gaits of bipeds. Walking requires that the duty factor of the each of the legs exceed 0.5 since, by definition,the feet must be on the ground simultaneously for a percentage of the gait cycle period. Lower duty factors (less than 0.5) result in ballistic motion identified with running, wherein the entire body leaves the ground for some duration.

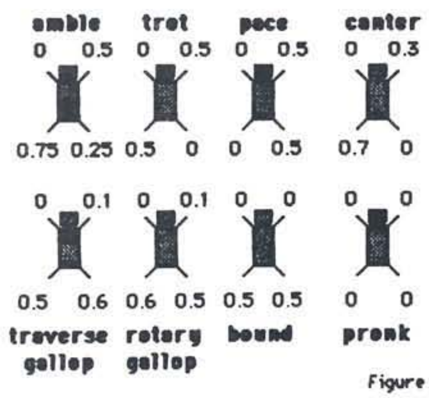
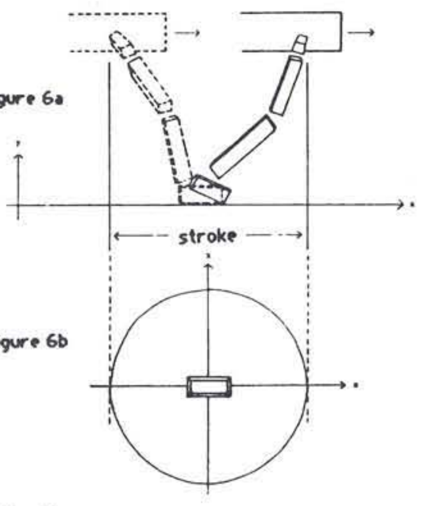


Figure 5

We will call the time a leg spends on the ground its *support duration*. The time spent in the air is the leg's *transfer duration*.

The *stroke* is defined as the distance traveled by the body during a leg's support duration. If we acknowledge that the foot must traverse the stroke during the transfer phase in order to "keep up" with the body, the stroke may alternatively be regarded as the length of the step taken by the leg over the ground (fig. 6a). The body may move over the ground plane in PODA, so the stroke in this context becomes the diameter of a circle in that plane (fig. 6b).



Leg Coordination

The following relationship holds between the legs and the body:

$$\text{supportDuration} = \frac{\text{stroke}}{\text{bodySpeed}} \tag{7}$$

The above equation solves for the time (or number of frames) that each leg must spend on the ground. By definition we also have

$$\text{dutyFactor} = \frac{\text{supportDuration}}{P}$$

The amount of time which a leg spends in the air depends on both the leg speed and the arclength of the transfer phase trajectory. That is:

$$\text{transferDuration} = \frac{\textbf{arcLength}(\text{transferTrajectory})}{\text{legSpeed}}$$

During the gait cycle period P, a single leg will move through one cycle of support and transfer, hence we have:

$$P = \text{supportDuration} + \text{transferDuration} \tag{8}$$

for any leg k. In fact, one may imagine the period as a duration subdivided into support and transfer durations (fig. 7). The *leg state* at time t may be determined as

$$\text{legState} = (\text{legState}_0 + t) \bmod P \tag{9}$$

where

$$\text{legState}_0 = (R_i)(P)$$

If the leg state is less than the support duration then the leg is in its support phase, otherwise the leg is in its transfer phase. Moreover, the time of foot placement occurs when the leg state equals zero and the foot liftoff occurs when the leg state is equal to the support duration (fig. 7).

An animator using PODA may design gaits for figures having any number of legs by instantiating the parameters given above. The model makes sure that all the variables are updated according to functional dependencies, thereby freeing the animator to experiment with the variables of interest such as relative phase without worrying about the integrity of the other related variables.

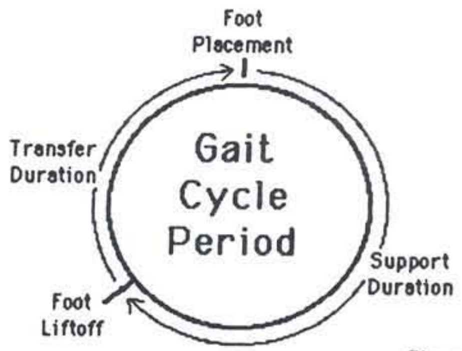


Figure 7

Leg Support and Transfer Trajectories

Aside from the problem of coordinating the timing of the legs, one must design the motion of the step taken during the transfer phase and insure that the feet will remain planted on the ground during the support phase.

A step is specified in PODA by the desired trajectory of the feet and the center angles and gains on the joints (which may change dynamically). The curve which defines the foot trajectory is defined by a Catmull-Rom (interpolating) spline. Because the desired shape of the curve depends on the geometry of the leg, the control points of the spline are set by moving the foot of the leg. The animator may conceptualize the design of the step as the specification of "key leg positions," in the spirit of a key-framing system. In PODA, a key position records the position of the foot (as a control point in the spline) and the center angles and gains that are associated with that position. The animator manipulates the foot into each position using PODA's inverse kinematic procedures, and then , once the foot is in place, the joint angles may be adjusted using the center angle and gain parameters.

This approach is distinguished from key-framing or joint angle interpolation systems in that the goal of achieving the desired foot position in Cartesian space is primary-the foot will travel precisely along the smooth Catmull-Rom spline from foothold to foothold. By contrast, if we interpolate the the leg positions in joint space, there is no general means of either moving the foot along a curve or placing the foot at a particular place on the ground.

The problem of keeping the legs on the ground as the body translates and rotates is simplified due to PODA's inverse kinematic capability. The problem reduces to solving for the position of the foot in the leg's moving coordinate system so that it is identical to the placement of the foot in the previous frame's world coordinate system (thereby keeping the foot stationary in the world). We we solve for this position using:

$$^{\text{World}}\overrightarrow{\text{prevFrameFoot}_{t-1}} = {}^{\text{World}}T_{\text{Hip}_{t-1}} \left({}^{\text{Hip}}\overrightarrow{\text{Foot}_{t-1}} \right) \tag{10}$$

$$^{\text{Hip}}\overrightarrow{\text{prevFrameFoot}_{t}} = {}^{\text{Hip}}T_{\text{World}_t} \left({}^{\text{World}}\overrightarrow{\text{prevFrameFoot}_{t-1}} \right)$$

Directional Control of the Body

If the animator is to have supervisory control over the legged figure, a means for directing the body's full translational and rotational degree's of freedom must be available. Given that all the legs are on the ground, the problem may be solved using equation (10). The fundamental problem is to calculate footholds and plan the foot transfer trajectories between them so as to adapt to the desired body motion.

Foothold and Transfer Trajectory Planning

An important concept of foothold planning is the notion of a *reference leg position*[22]. This is the desired position of the leg in mid-stance or half way though the leg's support duration (fig. 10). The posture of an animal when all of its legs are in their reference positions may be regarded as the "standing" position of the animal (fig. 11).

The other key ingredient for the foothold calculation is the ability to predict the body's future positions. In PODA, the body's trajectory may be computed as a function of the desired body trajectory over the ground plane (a cubic spline designed by the animator) and dynamic constraints due the timing and force limitations of legs (to be discussed) before the precise footholds are chosen. Since the body's position is known in advance, it is possible to plan ahead in order step toward the next stable position

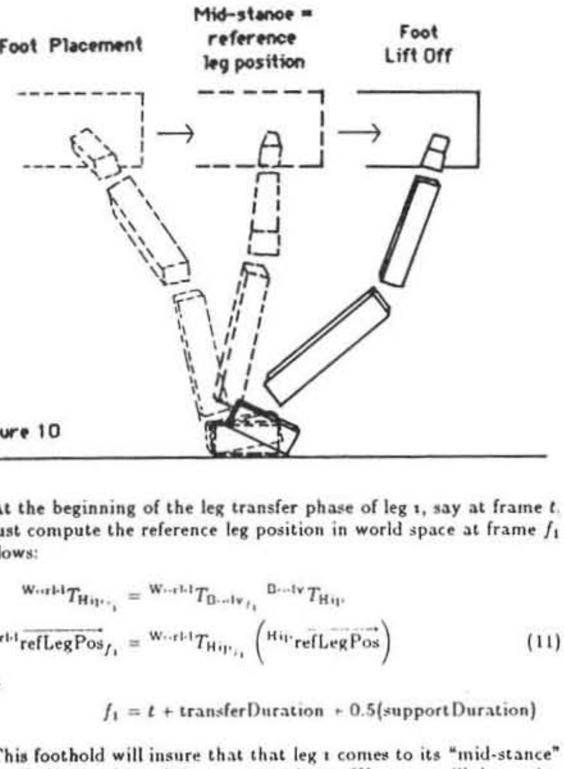


Figure 10

At the beginning of the leg transfer phase of leg i, say at frame t, we must compute the reference leg position in world space at frame f_1 as follows:

$$^{World}T_{Hip_{f_1}} = {}^{World}T_{Body_{f_1}} \; {}^{Body}T_{Hip}$$

$$^{World}\overrightarrow{\mathrm{refLegPos}}_{f_1} = {}^{World}T_{Hip_{f_1}} \left({}^{Hip}\overrightarrow{\mathrm{refLegPos}} \right) \qquad (11)$$

where

$$f_1 = t + \mathrm{transferDuration} + 0.5(\mathrm{supportDuration})$$

This foothold will insure that that leg i comes to its "mid-stance" position half way through its support phase. We must still determine the position of the foot in the body's coordinate system at the time the foot is placed down. This knowledge is required in order to facilitate moving the foot horizontally with respect to the body during the transfer. This may be accomplished by:

$$^{Body}\overrightarrow{\mathrm{refLegPos}}_{f_2} = {}^{Body}T_{World_{f_2}} \left({}^{World}\overrightarrow{\mathrm{refLegPos}}_{f_1} \right)$$

where

$$f_2 = t + \mathrm{transferDuration}$$

The generic transfer trajectory designed by the animator may then be adapted to move between the current foot position and the calculated foothold so that the height in the world and proportional distance moved next to the body are preserved.

Robotics research on walking vehicles has provided a rich source of computational models for the solution of body motion planning and leg coordination[22-26]. However, their design criteria is somewhat different the requirements of animation.

The primary design concerns of the robotics algorithms are to maintain dynamic stability of the walking vehicle, to avoid leg intersections, to optimize the load balancing and energy consumption, and to insure that the feet never stray beyond their kinematic limits. Because the algorithms must actually work for real walking machines (rather than simulated figures), their scope is conservative. Restrictions are placed on the types of gait patterns and relative phase relationships between the legs, thereby drastically limiting the repertoire of behaviors.

The design philosophy of PODA is to give the animator absolute control over the entire set of of available gaits in order to exploit the coupling between rhythm and dynamics (to be discussed) since these matters are of extreme importance in artistic design. Moreover, since PODA's current implementation on the Ridge 32C minicomputer provides for realtime computation of a figure possessing four 9-degree of freedom legs at 2 frames per second, leg interference and unnatural leg stretching may be detected immediately, leaving the range of many reasonable solutions to these problems up to the artist rather than hard-wiring a single solution into the motion model.

MODELING THE DYNAMICS OF LEGGED LOCOMOTION

The simulation of the dynamics of motion control in legged animals is an extremely complex modeling problem. Models for single limbs (industrial robots) which compute the relationships between the torques applied at the joints,the masses and moments of inertia of each of the links, and the position of the joints and their associated time derivatives, are well understood [38]. Work has also been published in the biomechanics field on the the relationship between muscular forces and motion parameters of simplified "ideal" models of animals[33,34]. Although these models may produce interesting animation, their appropriateness for artistic design and control must be considered as well as their (usually substantial) computational costs.

Simulation vs. Animation

In contrast to industrial robots and biomechanical simulations, animation does not necessarily require the computation of actual forces. The application of dynamics to animation is simplified by the fact that we are interested only in what can be seen.

The essential concern is to make the motion look as if forces were being applied. In other words, we are primarily interested in solving for the acceleration in dynamics models - the computation of parameters such as forces, torques and moments of inertia is only relevant if it can help us easily manipulate accelerations to produce coherent dynamic realism.

The necessity for modeling dynamics in PODA was apparent as soon as the kinematic model was completed. In a purely kinematic model the motion of the body is quickly seen to be independent from the coordination of the legs, and it appears as though the body is suspended from strings, pulling its legs behind it.

The development of dynamics for PODA is an ongoing research project. The initial goal was to see whether very simple dynamic models of legged locomotion could be developed which were both amenable to artistic control and as fully general as the kinematic model (applicable to any figure constructed by the animator). At the time of this writing, PODA is capable of modeling the translational acceleration of the center of mass of body in the vertical direction and ground plane, and the rotational acceleration of the body that is required to insure that it is facing in the direction of movement (if turning is desired). At all stages, the body's motion is constrained and propelled by the simulated forces applied by the legs.

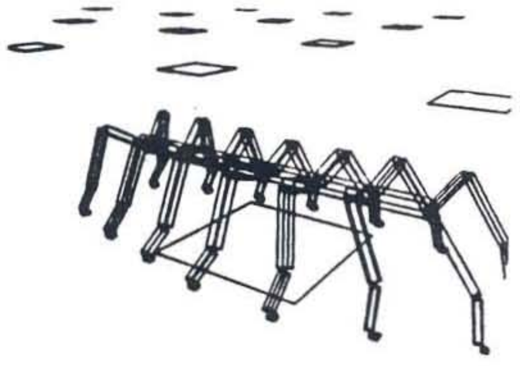
Decomposition of Dynamic Control in PODA

The simple model used in PODA was inspired by Raibert's work on legged hopping machines. He and his coworkers have built a one-legged hopping machine which is able to balance and move in three dimensions. His control algorithms are based on a decomposition into hopping height, forward velocity, and attitude control[31,28].

The model used in PODA decomposes the dynamic coupling between the legs and the body along two lines: decomposition by leg and decomposition by body direction.

Vertical Control

Dynamics in the vertical direction must take into account the effects of gravity and the gait cycle period of each leg. Since PODA's decomposition scheme is based on decomposition by leg, it will be helpful to consider a one-legged figure.


A 14 legged insect shown in its standing position.

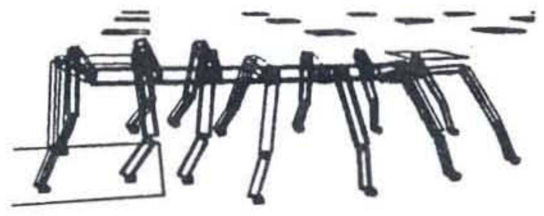

Moving in a wave gait, the legs near the rear advance toward their next footholds.

Figure 11

The current model makes the simplifying assumption that the upward force applied by the leg on the body is constant during its support phase. The animator supplies PODA with both the value of this force, the mass of the body, and the downward acceleration due to gravity. Net upward acceleration of the center of mass is then given by:

$$\vec{a}_{y_i} = \left(\frac{\vec{F}_{y_i}}{\text{mass}_{\text{body}}} - g \right) \text{ for leg } i \tag{12}$$

The gait cycle period may be subdivided into three dynamic stages of the leg's motion: pushing the body up, free falling, and then restoring the body to its original position (as long as the application of upward leg forces are symmetrical about the mid-stance position, the body will stabilize to zero velocity at that position). We will call these the *push duration*, *fall duration*, and *restore duration* respectively (fig. 12).

The leg support duration is a function of the body speed in the horizontal plane and the stroke (equation 7). Since the leg's traversal of the transfer trajectory must coincide with the body's ballistic motion, we have:

$$\text{transferDuration} = \left(\frac{\vec{a}_{y_i}}{g} \right) (\text{supportDuration}) \tag{13}$$

The vertical position of figures with multiple legs is determined in PODA by the superposition of the ballistic motion of the body due of each of its legs considered independently. This extremely simple model produces remarkably realistic motion for both walking and running in multiple leg figures: if the magnitude of vertical acceleration is low and the phase relationships of the legs are in opposition, the upward accelerations will cancel, resulting in a smooth walking oscillation. High accelerations resulting from strong single leg forces (e.g. running in a trot) or the sum of forces of many legs pushing from the ground together (e.g. hopping) propel the body into the air.

The trajectory taken by the body due to the summation of vertical leg accelerations and downward gravitational accelerations taken from each of the legs is automatically synchronized with rhythm of the phase relationships in the legs. For example, convincing cantors, trots, and bounding motion may be animated simply by altering the figure's gait.

Another advantage to PODA's leg decomposition of vertical dynamics is that changes in the figure's motion parameters which evolve over gait cycle periods, such as body speed or upward leg force may be easily accommodated by adjusting the related dynamic parameters for each leg's contribution independently at the beginning of its upward pushing phase. The stability of each leg's contribution guarantees the vertical stability of the body as a whole. A gait shifting algorithm has been developed by one of the authors which exploits the ability for legs to undergo phase shifts by varying the distribution of vertical pushing forces among them.

Horizontal Control

The desired horizontal path taken by the figure is specified in PODA by the animator with a cubic spline (Catmull-Rom or B-spline). Given the desired body speed along different parts of the curve, PODA may calculate the desired positions and velocities along it using a numerical arclength calculation.

However, a legged figure's acceleration toward a desired direction and speed must be coherent with its leg support duration pattern and it must also simulate the effects of momentum in a given direction in order to give the body a sense of weight.

In PODA, the body's ability to turn and speed up is consistent with the number of feet on the ground and the maginitude of the maximum achievable force $\|\overrightarrow{F_{\max_i}}\|$ assigned by the animator to each supporting leg. The maximum achievable acceleration of the body is governed by the sum of their forces:

$$\|\vec{a}_{xz_{\max}}\| = \sum_{i=1}^{n} \frac{\left\|\overrightarrow{F_{\max_i}}\right\|}{\text{mass}_{\text{body}}} \tag{14}$$

where n is the number of feet on the ground.

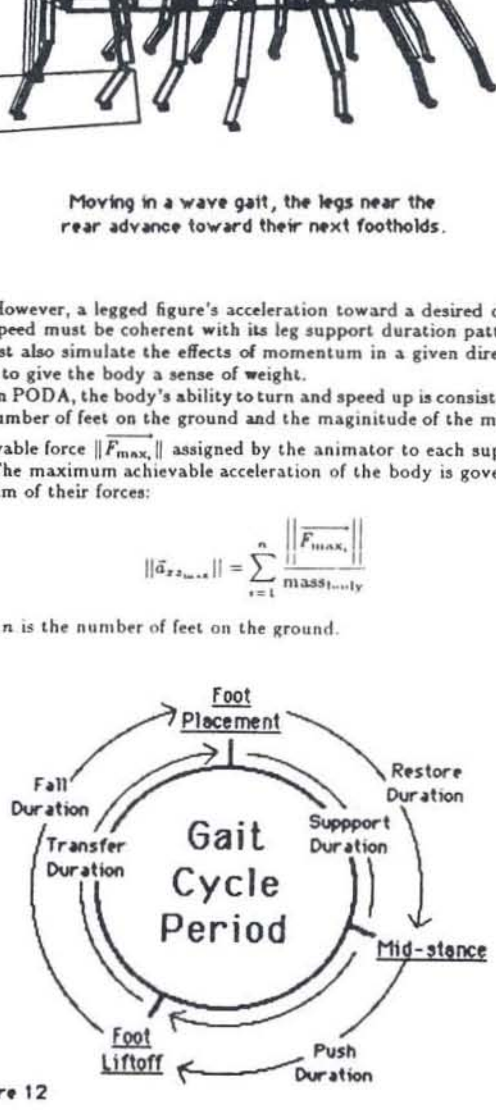


Figure 12

At each frame PODA computes the next desired velocity along the desired bouy path. Then PODA determines the desired acceleration at a given frame through velocity error feedback, that is, by subtracting the desired velocity from the current velocity at that frame. The horizontal acceleration of the body at frame t is then computed as:

$$\vec{a}_{xz} = \|\vec{a}_{xz}\| \frac{\vec{a}_{xz_{\text{desired}}}}{\|\vec{a}_{xz_{\text{desired}}}\|} \tag{15}$$

where

$$\|\vec{a}_{xz}\| = \min(\|\vec{a}_{xz_{\text{desired}}}\|, \|\vec{a}_{xz_{\max}}\|)$$

$$\vec{a}_{xz_{\text{desired}}} = \left(\dot{\text{Vel}}_{\text{desired}} - \dot{\text{Vel}} \right)$$

An additional backward acceleration during the restore duration and forward acceleration during the push duration is necessary in order to simulate the effects foot position with respect to the body at foot placement and foot liftoff [28]

Rotational Control

The foothold and leg transfer calculations described adapt to allow for full rotational control of the body. At the time of this writing, however, only the dynamics of accelerations about the yaw axis have been implemented (the pitch axis rotation apparent in figure; 2 is due to the separation of the application of vertical acceleration between the front and rear sections of the body). Yaw axis accelerations are necessary if we wish the body to realistically turn so that it is facing in the direction of movement.

The simple frame to frame strategy applied for horizontal control is not sufficient for coordinated turning, especially for running gaits wherein the legs are off the ground. In such cases one must know where the body is going in order to effectively anticipate the rotational accelerations required to keep the body properly oriented before its feet leave the ground.

The solution adopted by PODA for rotational control takes advantage of its ability to know the translational coordinates of the body in advance. The order of calculation for body dynamics plays an important role in this matter (fig. 13). The direction of body motion may be derived from the sequence of actual velocities taken by the body.

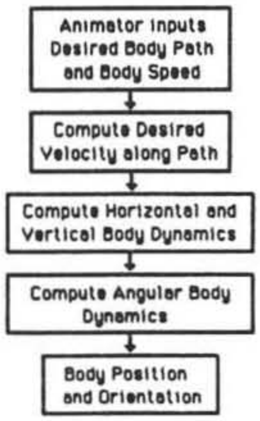

Animator Inputs Desired Body Path and Body Speed
Compute Desired Velocity along Path
Compute Horizontal and Vertical Body Dynamics
Compute Angular Body Dynamics
Body Position and Orientation

Figure 13

Once the desired body orientations are known, PODA is able to exploit the leg decomposition strategy employed for vertical control. Each leg applies a positive acceleration during its pushing phase and a negative restoring acceleration during its restoring phase in order to bring the body exactly to the desired angle computed at its mid-stance reference leg position. If we solve Newton's equations of motion with the constraint that we wish the final mid-stance rotational velocity to be zero, we have:

$$\ddot{\sigma} = \frac{(\sigma_{\text{midStance}+P} - \sigma_{\text{midStance}})}{(\text{pushDuration})(\text{pushDuration} + \text{fallDuration})}$$

where σ is angle about the yaw axis.

Conclusion

The described formulations have proven to be successful models for the synthesis of legged locomotion. However, many interesting problems remain to be solved. We will refine the legged locomotion model in PODA as more is learned from each simpler model. The addition of rotational dynamics for body pitch and, more generally, the modeling of body dynamics due to the motion of non-supporting limbs are obvious choices for extension.

Other problems of interest include the development of control techniques for maintaining postural balance, the inclusion of obstacle avoidance and collision detection, and a means of designing motor skills above and beyond the requirements of walking and running.

Acknowledgements

The authors wish to thank John Renner and Susan Amkraut for the preparation of this document, and Charles Csuri and Thomas Linehan for their maintenance of the stimulating environment at CGRG.

REFERENCES

[1] Ben-Israel, Adi and Thomas N. E. Greville, "Generalized Inverses: Theory and Applications," *Wiley-Interscience*, New York, 1974.

[2] Boullion, T. L. and P. L. Odell, "Generalized Inverse Matrices," *Wiley-Interscience*, New York, 1971.

[3] Denavit, J. and R. S. Hartenberg, "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," *ASME Journal of Applied Mechanics*, Vol. 23, pp. 215-221, June, 1955.

[4] Greville, T. N. E., "Some Applications of the Pseudoinverse of a Matrix," *SIAM Review*, Vol. 2, No. 1, January, 1960.

[5] Greville, T. N. E., "The Pseudoinverse of a Rectangular or Singular Matrix and its Applications to the Solutions of Systems of Linear Equations," *SIAM Review*, Vol. 1, No. 1, January, 1959.

[6] Hanson, R. J. and C. L. Lawson, "Extensions and Applications of the Householder Algorithm for Solving Linear Least Squares Problems," *Mathematics of Computation*, Vol. 23, pp. 787-812, 1969.

[7] Klein, C. A. and Huang, C. H., "Review of Pseudoinverse Control for Use with Kinematically Redundant Manipulators," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-13, No. 2, pp. 245-250, March/April, 1983.

[8] Liegeois, A., "Automatic Supervisory Control of the Configuration and Behavior of Multibody Mechanisms," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-7, No. 12, December, 1977.

[9] Maciejewski, A. A., and Klein, C. A., "Obstacle Avoidance for Kinematically Redundant Manipulators in Dynamically Varying Environments," to appear in *International Journal of Robotics Research*.

[10] Maciejewski, A. A., and Klein, C. A., "SAM - Animation Software for Simulating Articulated Motion," submitted to *IEEE Computer Graphics and Applications*.

[11] Noble, B., "Methods for Computing the Moore-Penrose Generalized Inverses, and Related Matters," pp. 245-301, in *Generalized Inverses and Applications*, ed. by M. A. Nashed, *Academic Press*, New York, 1975.

[12] Paul, R., *Robot Manipulators*, MIT Press, Cambridge, Mass., 1981.

[13] Penrose, R., "On Best Approximate Solutions of Linear Matrix Equations," *Proc. Cambridge Philos. Soc.*, Vol. 52, pp. 17-19, 1956.

[14] Peters, G. and J. H. Wilkinson, "The Least Squares Problem and Pseudo-Inverses," *The Computer Journal*, Vol. 13, No. 3, August, 1970.

[15] Ribble, E. A., "Synthesis of Human Skeletal Motion and the Design of a Special-Purpose Processor for Real-Time Animation of Human and Animal Figure Motion," Master's Thesis, The Ohio State University, June, 1982.

[16] Waldron, K. J., "Geometrically Based Manipulator Rate Control Algorithms", *Seventh Applied Mechanics Conference*, Kansas City, December, 1981.

[17] Waldron, K. J., "The Use of Motors in Spatial Kinematics," *Proceedings of the IFToMM Conference on Linkages and Computer Design Methods*, Bucharest, June, 1973, Vol. B, pp. 535-545.

[18] Whitney, D. E., "Resolved Motion Rate Control of Manipulators and Human Prostheses," *IEEE Transactions on Man-Machine systems*, Vol. MMS-10, No. 2, pp. 47-53, June, 1969.

[19] Whitney, D. E., "The Mathematics of Coordinated Control of Prostheses and Manipulators," *Journal of Dynamic Systems, Measurement, and Control, Transactions ASME*, Vol. 94, Series G, pp. 303-309, December, 1972, pp. 49-58.

[20] Alexander, R., "The Gaits of Bipedal and Quadrupedal Animals," *The International Journal of Robotics Research*, Vol. 3. No. 2, Summer 1984.

[21] McGhee, R.B., and Iswandhi, G.I., "Adaptive Locomotion of a Multilegged Robot over Rough Terrain", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-9, No. 4, April, 1979, pp. 176-182.

[22] Orin, D.E., "Supervisory Control of a Multilegged Robot", *International Journal of Robotics Research*, Vol. 1. No. 1, Spring, 1982, pp. 79-91.

[23] Klein, C.A., Olson, K.W., and Pugh, D.R., "Use of Force and Attitude Sensors for Locomotion of a Legged Vehicle over Irregular Terrain," *International Journal of Robotics*, Vol. 2, No. 2, Summer, 1983, pp. 3-17.

[24] Ozguner, F., Tsai, L.J., and McGhee, R.B., "Rough Terrain Locomotion by a Hexapod Robot Using a Binocular Ranging System," *Proceedings of First International Symposium of Robotics Research*, Bretton Woods, N.H., August 28, 1983.

[25] Lee, Wha-Joon, "A Computer Simulation Study of Omnidirectional Supervisory Control for Rough-Terrain Locomotion by a Multilegged Robot Vehicle," Ph.D. Dissertation, The Ohio State University, Columbus, Ohio, March, 1984.

[26] Yeh, S. "Locomotion of a Three-Legged Robot Over Structural Beams," Masters Thesis, The Ohio State University, Columbus, Ohio, March, 1984.

[27] Zeltzer, D.L., "Representation and Control of Three Dimensional Computer-Animated Figures," Ph.D Dissertation, The Ohio State University, Columbus, Ohio, March, 1984.

[28] Murphy, K.N., and Raibert, M.H., "Trotting and Bounding in a Planar Two-Legged Model," *Fifth CISM-IFTOMM Symposium on Theory and Practice of Robots and Manipulators*, June 26-29, 1984, Udine, Italy.

[29] Miura, H. and Shimoyama, I. "Dynamic Walk of a Biped," *The International Journal of Robotics Research*, Vol. 3, No. 2., Summer 1984, pp 60-74.

[30] Pearson, K.G., and Franklin, R., "Characteristics of Leg Movements and Patterns of Coordination in Locusts Walking on Rough Terrain," *The International Journal of Robotics Research*, Vol. 3, No. 2., Summer 1984, pp 101-107.

[31] Raibert, M.H., Brown, H.B.Jr, and Chepponis, M., "Experiments in Balance with a 3D One-Legged hopping Machine," *The International Journal of Robotics Research*, Vol. 3, No. 2., Summer 1984, pp 75-82.

[32] Korein, J.U., and Badler, N.I., "Techniques for Generating the Goal-Directed Motion of Articulated Structures," *IEEE Computer Graphics and Applications*, November 1982, pp. 71-81.

[33] Hemami, H. and Zheng, Y., "Dynamics and Control of Motion on the Ground and in the Air with Application to Biped Robots," *Journal of Robotic Systems*, Vol 1. No. 1, 1984, pp. 101-116.

[34] Hemami, H. and Chen, B. "Stability Analysis and Input Design of a Two-Link Planar Biped, *The International Journal of Robotics Research*, Vol. 3, No. 2., Summer 1984, pp. 93-100.

[35] McMahon, T.A., "Mechanics of Locomotion," *The International Journal of Robotics Research*, Vol. 3, No. 2., Summer 1984, pp 4-18.

[36] Lundin, R.V. "Motion Simulation," *Nicograph Proceedings 1984*, pp. 2-10.

[37] Orin, D. E., and Schrader, W. W., "Efficient Jacobian Determination for Robot Manipulators," *Sixth IFToMM Congress*, New Dehli, India, December 15-20, 1983.

[38] Orin, D.E., McGhee, R.B., Vukobratovic, M.,and Hartoch, G., "Kinematic and Kinetic Analysis of Open-Chain Linkages Utilizing Newton-Euler Methods," *Mathematical Biosciences*, Vol. 43, pp. 107-130, 1979.

Some of the equations in [Girard 85] do not reproduce well. Those equations are reprinted here.

(10)

$$^{\text{World}}\overline{\text{prevFrameFoot}}_{t-1} = {}^{\text{World}}\text{T}_{\text{Hip}_{t-1}} \left({}^{\text{Hip}}\overline{\text{Foot}}_{t-1} \right)$$

$$^{\text{Hip}}\overline{\text{prevFrameFoot}}_{t} = {}^{\text{Hip}}\text{T}_{\text{World}_t} \left({}^{\text{World}}\overline{\text{prevFrameFoot}}_{t-1} \right)$$

(11)

$$^{\text{World}}\text{T}_{\text{Hip}_{f\,1}} = {}^{\text{World}}\text{T}_{\text{body}_{f\,1}}\, {}^{\text{Body}}\text{T}_{\text{Hip}}$$

$$^{\text{World}}\overline{\text{refLegPos}}_{f\,1} = {}^{\text{World}}\text{T}_{\text{hip}_{f\,1}} \left({}^{\text{Hip}}\overline{\text{refLegPos}} \right)$$

(and lower on that same page)

$$^{\text{Body}}\overline{\text{refLegPos}}_{f\,2} = {}^{\text{Body}}\text{T}_{\text{World}_{f\,2}} \left({}^{\text{World}}\overline{\text{refLegPos}}_{f\,1} \right)$$

(12)

$$\overline{a}_{y_i} = \left(\frac{\overline{\overline{\text{F}}}_{y_i}}{\text{mass}_{\text{body}}} - \text{g} \right) \text{ for leg } i$$

(13)

$$\text{transferDuration} = \left(\frac{\overline{a}_{y_i}}{\text{g}} \right) \text{(supportDuration)}$$

(14)

$$\left|\left|\overline{a}_{xz_{\max}}\right|\right| = \sum_{i=1}^{n} \frac{\left|\left|\overline{\mathrm{F}_{\max_i}}\right|\right|}{\mathrm{mass}_{\mathrm{body}}}$$

(15)

$$\overline{a}_{xz} = \left|\left|\overline{a}_{xz}\right|\right| \frac{\overline{a}_{xz_{\mathrm{desired}}}}{\left|\left|\overline{a}_{xz_{\mathrm{desired}}}\right|\right|}$$

where

$$\left|\left|\overline{a}_{xz}\right|\right| = \min\left(\left|\left|\overline{a}_{xz_{\mathrm{desired}}}\right|\right| , \left|\left|\overline{a}_{xz_{\max}}\right|\right| \right)$$

$$\overline{a}_{xz_{\mathrm{desired}}} = \left(\overline{\mathrm{Vel}}_{\mathrm{desired}} - \overline{\mathrm{Vel}} \right)$$

(On page 269)

$$\ddot{\sigma} = \frac{\left(\sigma_{\mathrm{midstance+P}} - \sigma_{\mathrm{midstance}} \right)}{\left(\mathrm{pushDuration} \right) \left(\mathrm{pushDuration} + \mathrm{fallDuration} \right)}$$

Applying Classical Techniques to Computer Animation

Glenn McQueen
Computer Graphics Laboratory
New York Institute of Technology
Old Westbury, New York 11568

Introduction

This paper discusses principles of conventional animation and how they can enhance the quality of computer generated animation. Since these tricks improve the look of any animation and take no longer to render than unembellished animation, these principles are well suited for animation in a production environment.

Principles of Animation

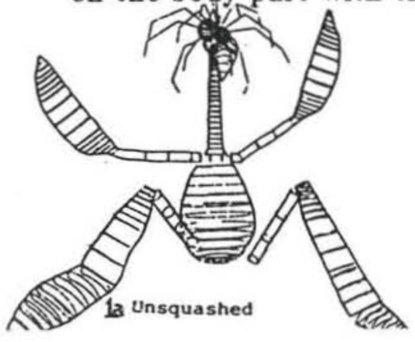
It may be more accurate to refer to these tricks not as principles of animation, but as principles of communication. Anyone can animate - just move a group of objects around and you've got animation. But animating successfully implies much more than moving stuff from point A to point B. Effective animation elicits a response from the viewer, it captivates and entertains [Thomas81]. Character animation is well suited for the development of an emotional link between the audience and the animation. Flying logos, on the other hand, are sometimes devoid of the endearing qualities which allow an audience to empathize with a cartoon character. How much emotional appeal can be squeezed out of a ten second fly-through? If a storyboard is presented which will allow the animator the freedom to apply some character to a logo, terrific. If not, then the difficult job of 'captivating and entertaining' without the benefit of a character lies ahead.

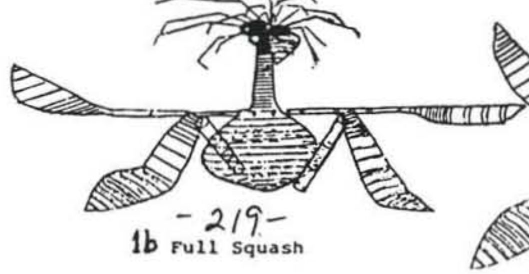
Think of an animator as a sleight-of-hand artist. He has a selection of tricks which fool the eye into seeing something which may or may not be there [Luckliesh65]. Perhaps the most basic 'trick' is to observe the motion of an object, see the essence of its movement, and determine the best way to simulate it. From there we add embellishments to jazz up the motion. *Squash* and *stretch* is one of these embellishments. This emphasizes the apparent elastic, rubbery properties of a shape, for example when a cartoon cat falls from a third story window. Its body stretches on the way down and then squashes as it bounces into the dog catcher truck. *Overlapping action* refers to the parts of a character which start or stop moving at different times, as in the cat's head remaining on the screen a few frames longer than the falling body, and then snapping down to catch up. *Secondary motion* addresses minor action which supports the main action in a scene, as in garbage cans being knocked over on the landing. *Staging* deals with the overall composition of a scene, while *arcs* refer to the principle that most actions aren't straight, but follow some sort of curve. The principles of *ease in ease out* allow for smooth acceleration and deceleration of motion. *Exaggeration* exaggerates the essential elements of a movement, drawing them to the audience's attention. *Anticipation* refers to movement just before a major action which allows the audience to anticipate the coming action, as in the cat scrambling in air for a few seconds before falling. Finally the principle of *timing and pacing*, which refers to how timing influences the overall effect of a piece of animation.

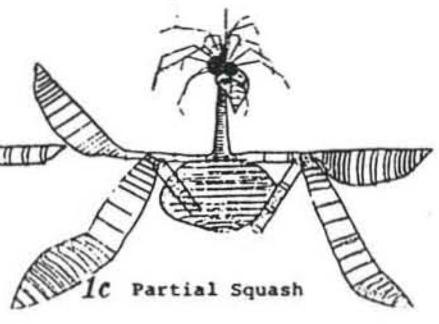
An understanding of these principles is important, so let's go over them in more detail.

Squash and Stretch

Perhaps the easiest way to breathe life into a static piece of animation is to incorporate *squash* and *stretch* If a character or object is in motion it will undergo certain changes within its overall shape. As in our example of the cat, a character falling through space stretches in the direction of the fall, and squashes or 'splats' when it hits the ground. The scaling may seem extreme when viewed in a single frame, but in motion it is remarkable how much the squashing and stretching can be exaggerated while still retaining a natural look. This elasticity can be used to imply weight, mass or other physical qualities. For example, the shape of an iron balloon would not be affected by a drop to the ground, while a balloon full of cooking oil would go through drastic shape changes both as it dropped and when it hit the ground. Computer graphics easily supports global axial scales, allowing the animator to 'keep-the-volume-of-the-object-constant' throughout the scene. If the sphere is scaled by a .5 unit squash along its length to imply a squishy ball smacking against the ground, an appropriate scale, say 1.5, units affecting its width would be necessary to keep the implied volume constant. As usual, more complex models present more complex problems. If a hierarchically defined character lands with a thud on his butt, a global scale in Y would not be appropriate, as this would also squish his legs, feet and everything else (*fig. 1b*). This implies equal weight or mass among all his parts, when in fact his torso and neck would be squishing the most. Scaling the peripheral body parts back up by a percentage of the original scale keeps the visual weight on the body part with the most implied mass (*fig. 1c*).


1a Unsquashed


1b Full Squash


1c Partial Squash

Ideally, a flexible model would be used, in which the shape of various parts can be appropriately changed. This would allow the character to take on much a pear-shaped squash, more convincing in implying weight.

Overlapping Action, Follow Through

If all the parts of a character stop or change motion at the same time, the effect is one of extreme rigidity. To impart a sense of fluidity, animators delay the movement of appendages. In a keyframe animation system, keys can be set to block out the main motion of the character. Once the animation has been previewed and approved, *overlapping action* can be added. For example let's look at a piece of animation in which a character falls on the ground. Assume that the main motion has been blocked out. Now, move the last frame (the one where he hits the ground) forward a couple of frames. You now have two identical 'last frames', one slightly after the next. Go back to the original 'ground-hitting' frame and move the character's arms and hair back up to a 'falling' position. Interpolate the 'between' frames. The sequence of frames now shows the character landing, with its arms and hair coming to rest a few frames later (*fig. 2*).

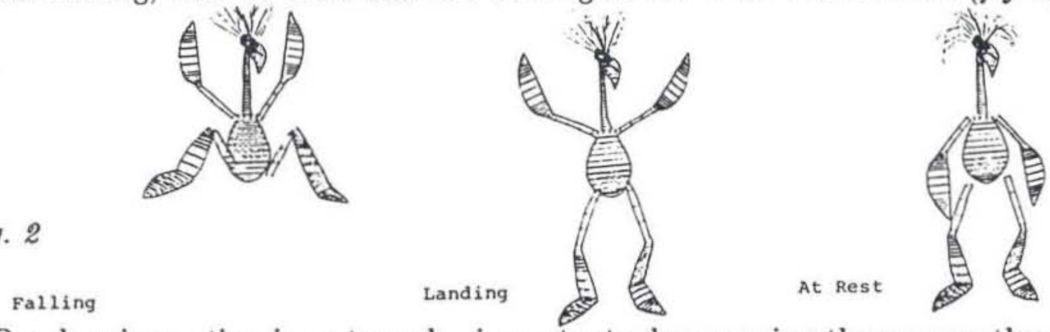


fig. 2

Overlapping action is extremely important when moving the camera through an environment or zooming a logo through space. Early computer animation typically comprised of a move, a pause, a rotation, a pause, another move, another pause, etc. This quickly becomes tedious. A solution is to start the rotation before the move finishes, substituting the pauses for overlapping action. *Follow-*

through is a common form of overlapping action. Rather than abruptly stopping an action after it has been completed, the motion eases out along the same path of action. For example, a tennis swing is much more effective if the swing continues after the ball has been hit. Follow-through gives continuity and fluidity to movement.

Ease In, Ease Out

A judicious use of *eases* can greatly enhance the look of even the most basic move or translation. An ease is a gradual speeding up or slowing down of a motion usually used at the beginning or end of an action to 'soften' the transition between active and static. Many animation systems give a wide choice of eases, the most common being linear or exponential. Linear eases slow in and out at the same rate, so all motion proceeds in a steady, predictable manner. They do not particularly lend themselves to interesting animation and are least desirable. Exponential eases are more widely used, as their motion is fluid and more enjoyable to watch. Being able to actually define your own ease is the best alternative. At NYIT, a curve editor is used to interactively edit curves which represent x, y and z transformations. This system is ideal for visualizing eases, allowing both linear and 'hand made' exponential eases. A splining function, when used in conjunction with a relaxing algorithm allows for an infinite number of possible eases. There are accepted algorithms defining the mathematical specifications of all the myriad eases, but actually

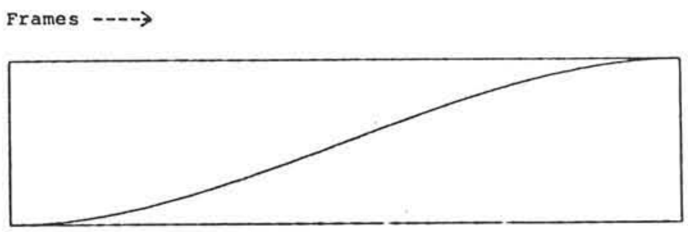


fig. 3

seeing the curve dip down to its rest position is sometimes as useful as seeing the animation in preview (*fig. 3*). The ability to interactively adjust a curve which determines the rate of animation or transition between positions in a keyframe is crucial. The flexibility of interactively adjustable curves cannot be stressed enough, as each job requires a slightly different move and tries to give a different feel. A standard ease might not give the animator the flexibility he needs to impart a sense of mass to, say, a heavy object ; a really slow ease-in, while an hand drawn ease might.

Exaggeration

Classically trained animators often speak of using *exaggeration* to 'sell' the action or the movement of a character. This is not meant to imply that exaggeration of motion is always the way to go, but often exaggerating certain motion characteristics often is necessary to maintain a degree of interesting motion. Exaggeration doesn't have to impart a cartoony feel to the animation to be effective. For example, if a client requests a piece of animation showing a computer generated dancer completing a pirouette, source footage could be used but only up to a point. After the motion has been blocked out, it's up to the animator to decide which movements of the dancer must be exaggerated in order to enhance the animation. Kenneth Wesley's skater animation for "CBS Sports Saturday [1986]" is a terrific example. To assist the animator, the client provided live action footage for reference. The live action was used to rough out the major movements, which were then subtlely exaggerated to showcase the graceful aspect of the skater. The effect is neither cartoony nor stiff, but natural and fluid.

Anticipation

One of the oldest theatrical devices used to improve the readability of animated shorts is *anticipation.* Anticipation occurs when a major action is preceded by a specific move which allows the audience to predict what is about to happen. This anticipation can be as small as a change of expression or as big as a broad physical action. In (*fig. 4*) the character gathers itself up like a spring, drawing back in the opposite direction before zooming off.

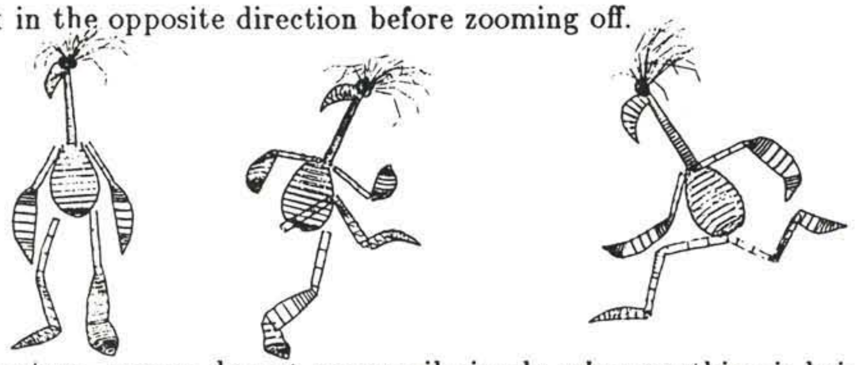


fig. 4

These anticipatory moves do not necessarily imply *why* something is being done, but make clear *what* is being done. Although anticipation works best when dealing with characters, the practice of cluing the audience in to what is about to happen is valid for more straightforward animation as well. Once a move has been implied through anticipation, animating a vastly different move can be used to introduce an element of surprise. For example, a car coiling up, obviously ready to shoot forward but then zooming backward could be considered a sight gag. Of course, this kind of slapstick may not suit all tastes or all production situations.

Secondary Action

Secondary action refers to additional motion which helps support the main action in a scene. This particular type of motion is usually subtle, and occurs either before or after the main action so as not to compete with the primary movement. For example, a character shuffling his feet slightly while waiting for someone reinforces his uneasiness, giving the audience the visual cues they need to understand his character.

In real life, people or animate objects seldom stand perfectly still - there's always some secondary motion. Someone sitting might tap their toes or drum their fingers against the arm of a chair, while a plant might sway slightly in a breeze. Without this subtle secondary motion, animation takes on a sterile, robotic look.

Like overlapping action, anticipation and squash and stretch, secondary action is best added after the main action has been roughed out.

Arcs

Most motion is not linear, but describes some sort of *arc*. This basic premise is based on classical character animation, but has obvious applications to the field of computer animation. Rather than linearly interpolating from one keyframe to the next, passing a curve through the keys gives a more dynamic look to the animation. If animation has been completely interpolated using splines, however, the motion may be too smooth, too uniform in velocity - in short, it will have no punch. Any 'ooomph' lost by splining can be regained by editing curves by hand. Again, a function editor which gives an interactive graphic representation is ideal for defining motion curves. Most systems have some automatic interpolation functions available to the animator. At NYIT we have the choice between user defined, linear, or cubic interpolating splines. One problem with cubic interpolating splines is that in their quest to keep slope continuity from keyframe to keyframe they tend to overshoot when confronted with sudden changes in velocity [Heckbert85]. Since animators intend keyframes to represent extremes in motion, these overshoots can have disastrous results. Feet go through the floor, fingers go

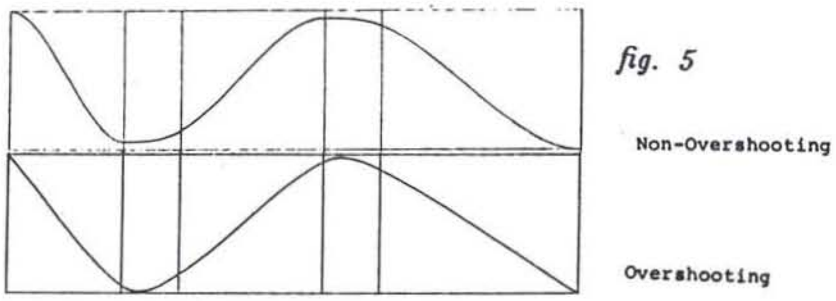


fig. 5

through hands, dogs sleep with cats. In short, chaos ensues. Non-overshooting cubic interpolating splines (*fig. 5*) are necessary in a production animation environment as they allow smooth motion without jumping in and out of a curve editor.

Staging

Perhaps the most basic trick of all, *staging*, refers to the total composition of a piece of animation. Even if a scene is well animated, the desired effect could be lost if the scene isn't well set up. Every cut, every shot, should reflect and reinforce the theme or mood in the storyboard. An action should be staged so that it is unmistakably clear, a mood staged so that it has the most impact on the audience. A good animation system should give the animator an equivalent degree of degree of scene control afforded to a live action director.

Some systems offer several cameras, most of which can be gimballed, dollied, moved and rotated, while others give the option of changing not only the camera position but where the camera is looking as well. Being able to change *only* the 'point of interest' and camera position is useful for plotted moves and fly-throughs, but can be cumbersome when a simple pan or dolly is called for. Here camera orientation is a more useful control.

Being able to choose from a number of cameras helps in coming up with the most effective shot. Multi-camera systems make the animation process easier, as well. If complex motion between several objects is required, several well placed cameras help alert the animator to unwanted intersections, preventing nasty surprises during the rendering process. Most animation systems have not only an infinite choice of camera positions, but an infinite choice of focal lengths as well. A slight lens change can make a dramatic difference in the look of a piece of animation.

Timing / Pacing

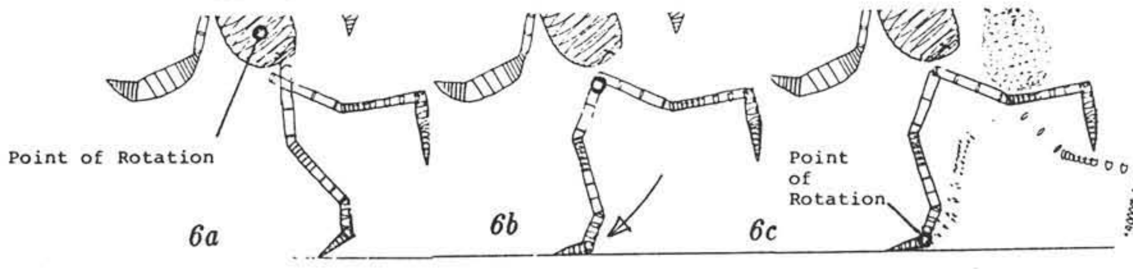
Timing can be one of the most frustrating aspects of production oriented computer animation. How many times have you laboured for weeks over models, lighting and texturing, only to find that the client has cut the animation from eleven seconds down to seven? Usually a client will want a great deal of action in a very short period of time. If he was dealing with live action chances are he would be a little more lenient with his stopwatch, but the cost-per-second of computer animation makes every moment count. Limited budgets are partially to blame for some of the too-fast fly throughs seen every night on TV. The timeframe usually allotted to TV spots is five to twenty seconds, with an occasional thirty second piece. These time constraints, when coupled with tight delivery/air-date deadlines allow for few animation changes during a production. Sometimes a persuasive animator can squeeze a couple more frames out of a client, permitting a reasonable ease-in or a graceful camera move, but a couple of frames is usually the limit. A useful tool for handling these situations is one that allows the animation to be expanded or contracted to different lengths without affecting the motion or the relative timing of the piece. This expansion or contraction also could be applied to just sections of the animation, changing the timing of the main action while leaving the eases at each end untouched.

Computers in Particular ...

Although the aforementioned principles are of great help when producing animation, they do not address many animation problems. Facial animation, interaction between models and environments, clothing, even simple walk cycles present unique problems which may require specialized software solutions.

Tree Traversal

One difficulty associated with conventional hierarchically defined models is the inflexibility of their tree structure. For example, as the torso is rotated forward to begin the fall into a walk cycle, both legs rotate off the floor (*fig. 6a*). The brute force solution to this problem is to rotate each leg back to the desired location for each keyframe (*fig. 6b*). A more elegant solution is to re-root the hierarchy of the model, allowing the animator to change root nodes on the fly. This would allow the animator to change the root node and pivot the model around the right foot for one step (*fig. 6c*), then change the root node, allowing the next step to be pivoted around the left foot. This implies two distinct choices of rotation, a bend which will root the tree at the torso and a pivot, which will root the tree at the appropriate end node.



Inverse Kinematics

Inverse kinematic chains have several uses for character animation. Let's use a walk cycle as an example. By setting the feet at their desired position and invoking inverse kinematics, the rotations for the ankles, knees and hips are derived. Although this keeps the feet on the floor, the combination of rotations which inverse kinematics give may not be the ones the animator desires. The ability to create goal-oriented keyframes is important, but so is control over the methods used to attain the goal.

Parameterized Systems

Parameterized animation allows the animator to apply scales, rotations, transformations or other operations to a series of points through a single parameter. Both the number of points affected and their interelationship is included in a script describing each parameter. Let's use facial animation as an example. Facial animation lends itself to parameterized systems since different parts of a body can be addressed separately, unlike a face which has no distinct delineation between elements. Several successful attempts at facial animation have been made by [Parke82], and later at NYIT utilizing the program, EM [Hanrahan85]. EM allows the animator to use script equations to control input parameters, which can then be manipulated on an E&S calligraphic system using a set of dials. The parameter 'Rsmile' might affect the vertex at the right corner of the mouth on a 1*1 ratio, while affecting the immediately surrounding vertices on a 1*.5 ratio. This means if the animator applies a transformation of .5 units to 'Rsmile', the corner vertex will move .5 units, while the surrounding vertices in 'Rsmile' will move .25 units (*fig. 7*). The success of this system greatly depends on the skill with which the parameters are described, but once a flexible set of parameters is created the model can be manipulated with a tremendous degree of precision and freedom.

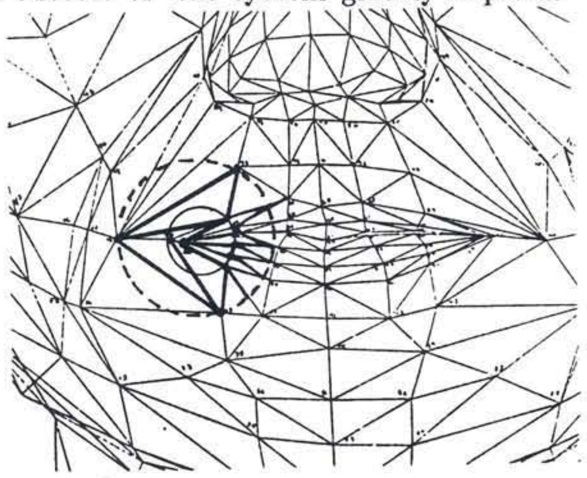


fig. 7

Programmed Animation

Occassionally a piece of animation is just too complex to be animated by manually positioning and requires a specialized program to animate the objects. For example, if the storyboard called for thousands of little tiles to randomly fly around then suddenly form a sphere, it might make more sense to write a short program to control their motion rather than spending hours animating them with an interactive system. Although simple random motion generators may not give the animator the control he would prefer, on occasion they provide exactly what's needed. Another situation which would lend itself to programmed animation would be a vehicle driving over an uneven landscape [Lundin84]. Keyframe interpolation is tedious in this case, as the changing landscape requires keys at each frame.

Conclusion

Over the last few years many advances have been made towards generating higher quality animation. Rendering techniques such as ray tracing and temporal anti-aliasing have added to animation's illusion of reality, but require tremendous amounts of computing power as well as a certain technical enlightenment on the part of the user.

Classical animation techniques, on the other hand, have been in use successfully for decades and can be used on systems ranging from Crays to Commodores. Comprehensive animation systems which allow animators the freedom afforded to classical animators will improve the art of computer animation *without* enormous computational power. By implementing principles of classical animation we can approach the challenge of creating terrific computer generated animation from a less technically-oriented direction. Great computer animation will always be a mixture of programming wizardry and the black art of classical character animation.

References

[Heckbert85] Heckbert, Paul. *Non-Overshooting Hermite Cubic Splines for Keyframe Animation.* New York Institute of Technology Computer Graphics Laboratory 3D Technical Memo #10, February 7 1985

[Hanrahan85] Hanrahan, Pat and Sturman, David. *Interactive Animation of Parametric Models.* Course Notes; Introduction to Computer Animation ACM SIGGRAPH 1985.

[Luckliesh65] Luckliesh, M. *Visual Illusions, Their Causes, Characteristics & Applications.* Dover Publications, 1965.

[Lundin84] Lundin, Richard V. Motion Simulation. *Nicograph 1984 Conference Proceedings.* Nicograph, November 1984.

[Parke82] Parke, Frederick I. Parameterized Models for Facial Animation. *IEEE Computer Graphics and Applications*,Nov 1982.

[Thomas81] Thomas, Frank and Johnstone, Ollie. *Disney Animation, the Illusion of Life.* Abbeville Press, New York, 1981.

Suggested Reading

1. Blair, Preston. *Animation.* Walter T. Foster Art Books, Tustin, CA, 1949.

2. Blair, Preston. *How to Animate Film Cartoons.* Walter T. Foster Art Books, Tustin, CA, 1980.

3. Halas John, Manvell, Roger. *The Technique of Film Animation.* Hastings House ,1976.

4. Halas, John. *Computer Animation.* Hastings House, 1974.

5. Sturman, David. *A Discussion on the Development of Motion Control Systems.*Course Notes; 3d Motion Specification and Control, ACM SIGGRAPH 1986.

AN INDEXED BIBLIOGRAPHY ON COMPUTER ANIMATION

N. Magnenat-Thalmann and D. Thalmann
MIRA Lab. HEC/IRO
University of Montreal
Montreal, Canada
1985

Although the computer plays an ever-increasing role in animation, the term "computer animation" is imprecise and can sometimes be misleading. This is because the computer can play a variety of different roles. A popular and simple way of classifying animation systems is to distinguish between computer-assisted and modelled animation.

Computer-assisted animation consists mainly of assisting conventional animation by computer. In particular, the computer can be used to input drawings, to produce in-betweens, to specify the motion of an object along a path, to color the drawings and create a background, to synchronize motion with sound and to initiate the recording of a sequence on film. This type of computer animation is also mainly carried out in two dimensions.

With modelled animation, the computer becomes more than a support, playing a basic role in the creation of a three-dimentional world. This type of computer animation involves three main activities: object modelling, motion specification and synchronization and image rendering.

A "computer animation" bibliography could include references on a wide variety of related subjects: e.g. graphics editing, computer-aided geometry, computer art and image synthesis. Based on John Halas statement that "movement is the essence of animation", the authors decided to focus on the references on motion in two and three dimensions. This bibligraphy, therefore, contains an exhaustive list of papers on key-frame systems and computer animation systems and languages. Papers on the fast developing topic of human modelling and animation are also listed. Although object modelling is an important part of modelled computer animation, we have not listed the numerous papers on this subject. Readers are invited to consult the very completed bibliography on

This bibliography also appeared in
"IEEE Computer Graphics & Applications", July 1985

computer-aided geometric design by Brian Barsky (IEEE Computer Graphics and Applications,July 1981,pp.67-109). Image rendering is also an important process in the production of computer-generated films. However, image synthesis is a research domain at the same level as computer animation, so we have only listed papers that describe techniques directly linked to animation like particle systems or motion blur. Most important papers on image synthesis may be found in the SIGGRAPH proceedings, IEEE Computer Graphics and Applications and ACM Transactions on Graphics.

Papers listed in this bibliography are mainly research papers; although we cannot guarantee that this list contains no errors or omissions, we believe it to be accurate and complete. We have also listed several papers intended for a wider audience that we judge to be of significant interest.

The index allows the reader to find all the papers related to a theme or belonging to a class. The same paper may be present in more than one class. For example, a paper which discusses a computer animation system for animating human bodies using in-betweens will be classified in three categories: computer animation systems, key-frame animation and human modelling and animation.

A

1. Ackland, B. and Weste, N. (1980) "Real Time Animation Playback on a Frame Store Display System", Proc. SIGGRAPH'80, Computer Graphics, Vol. 14, No 3, pp. 182-188.

2. Alexander, S. and Huggins, W.H. (1967), User's Manual on PMACRO, John Hopkins University.

3. Armbrust, R. (1983) "The Simulation of Space", Computer Pictures, Vol. 1, No 1, pp. 24-27.

4. Armstrong W.W. and Green M. (1985) "The Dynamics of Articulated Rigid Bodies for Purposes of Animation", Proc. Graphics Interface '85, Montreal.

5. Auger, R. (1983) "3D Computer Animation in Television Advertising", Proc. CG'83, London, pp. 259-266.

B

6. Badler, N.I. (1975) Temporal Scene Analysis: Conceptual Descriptions of Objects Movements, Ph.D. Diss., Univ. of Toronto.

7. Badler, N.I. (1982) "Human Body Models and Animation", IEEE Computer Graphics and Applications, Vol. 2, No 9, pp. 6-7.

8. Badler, N.I. (1984) "What is Required for Effective Human Figure Animation?", Proc. Graphics Interface '84, Ottawa, pp. 119-120.

9. Badler, N.I. and Morris, M.A. (1982) "Modelling Flexible Articulated Objects", Proc. Computer Graphics '82, Online Conf., pp. 305-314.

10. Badler, N.I., O'Rourke, J. and Kaufman, B. (1980) "Special Problems in Human Movement Simulation", Proc. SIGGRAPH'80, Computer Graphics, Vol. 14, No 3, pp. 189-197.

11. Badler, N.I., O'Rourke, J. and Toltzis (1979) "A Spherical Representation of a Human Body for Visualizing Movement", Proc. IEEE, Vol. 67, No 10, pp. 1397-1403.

12. Badler, N.I. and Smoliar, S.W. (1979) "Digital Representations of Human Movement", Computing Surveys, Vol. 11, No 1, pp. 19-38.

13. Baecker, R.M. (1969) Interactive Computer-Mediated Animation, Ph.D. Dissertation, MIT, Project Mac-Tr-61.

14. Baecker, R.M. (1969) "Picture-driven Animation", Proc. Spring Joint Computer Conference", AFIPS Press, Vol. 34, pp. 273-288.

15. Baecker, R. (1970) "Current Issues in Interactive Computer-Mediated Animation", Proc. 9th Annual Meeting UAIDE, pp. 273-288.

16. Baecker, R.M. (1971) "From the Animated Student to the Animatéd Computer to the Animated Film to the Animated Student...," Proc. Purdue 1971 Symp. Appl. Comput. Electr. Eng. Educ., Purdue University.

17. Baecker, R.M. (1976) "A Conversational Extensible System for the Animation of Shaded Images", Proc. SIGGRAPH'76, Computer Graphics, Vol. 10, No 2, pp. 32-39.

18. Balchin, N. (1983) "Film Animation by Microcomputer", Proc. CG'83, London, pp. 197-204.

19. Begley, S. (1982) "The Creative Computers", Newsweek, July 12, pp. 44-47.

20. Booth, S., Kochanek, D.H. and Wein M. (1983) "Computers animate film and video", IEEE spectrum, pp. 44-51.

21. Booth, K.S. and MacKay, S. (1982) "Techniques for Frame Buffer Animation", Proc. Graphics Interface '82, pp. 213-219.

22. Borrell, J. (1981) "The Magic of Computer Animation", Computer Graphics World, No 10, pp. 25-33.

23. Bosche C. (1967) "Computer Generated Random Dot Images", Design and Planning, 2, pp. 87-92.

24. Brown, M.H. and Sedgwick, R. (1984) "A System for Algorithm Animation", Proc. SIGGRAPH'84, pp. 187-194.

25. Burtnyk, N., Pulfer, J.K. and Wein, M. (1971), "Computer Graphics and Film Animation", INFOR, pp. 1-11.

26. Burtnyk, N. and Wein M. (1971) "Computer-Generated Key-frame Animation", Journal of Society for Motion Picture and Television Engineers, Vol. 80, pp. 149-153.

27. Burtnyk, N. and Wein, M. (1971) "A Computer Animation System for the Animator", Proc. UAIDE 10th Annual Meeting, pp. 3-5 to 3-24.

28. Burtnyk, N. and Wein, M. (1974) "Towards a Computer Animating Production Tool", Proceedings Eurocomp Congress, Online, Brunel, England, 1974, pp. 174-185.

29. Burtnyk, N. and Wein, M. (1976) "Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key Frame Animation", Comm. ACM, Vol. 19, No 10, pp. 564-569.

30. Buxton, W. (1982) "Computer Assisted Filmmaking", American Cinematographer, Vol. 63, No 8.

C

31. Calvert, T.W. and Chapman, J. (1978) "Notation of Movement with Computer Assistance", Proc. ACM Annual Conf., Vol. 2, pp. 731-736.

32. Calvert, T.W., Chapman, J. and Patla, A. (1980), "The Integration of Subjective and Objective Data in Animation of Human Movement", Proc. SIGGRAPH'80, Computer Graphics, Vol. 14, No 3, pp. 198-203.

33. Calvert, T.W., Chapman, J. and Patla, A. (1982) "Aspects of the Kinematic Simulation of Human Movement", IEEE Computer Graphics and Applications, Vol. 2, No 9, pp. 41-50.

34. Calvert, T.W., Chapman, J. and Patla, A. (1982b) "The Simulation of Human Movement", Proc. Graphics Interface '82, pp. 227-234.

35. Carr, J.W., et al., (1970) "Interactive Movie Making", Proceedings 9th UAIDE Annual Meeting, pp. 381-397.

36. Catmull, E. (1972) "A System for Computer Generated Movies", Proc. ACM Annual Conference, pp. 422-431.

37. Catmull, E. (1978) "The Problems of Computer-assisted Animation", Computer Graphics, Vol. 12, No 3, pp. 348-353.

38. Catmull E. (1979) "New Frontiers in Computer Animation", American Cinematographer, October Issue.

39. CGW (1982) "Digital Paint Systems Survey", Computer Graphics World, Vol. 5, No 4, pp. 62-65.

40. Christopher, R. (1982) "Digital Animation does Dallas", Videography, February Issue, pp. 37-42.

41. Chuang, R. and Entis, G. (1983) "3D Shaded Computer Animation - Step by Step", IEEE Computer Graphics and Applications, Vol. 3, No 9, pp. 18-25.

42. Citron, J. and Whitney, J. (1968) "CAMP Computer Assisted Movie Production", FJCC, AFIPS Conference Proceedings, Vol. 33(2), pp. 1299-1305.

43. Colonna, J.F. (1983) "From Display of Computer Results to Artistic Creation", Proc. CG'83, London, pp. 219-232.

44. Crow, F.C. (1978) "Shaded Computer Graphics in the Entertainment Industry", Computer, IEEE Press, 11 (3), pp. 11-23.

45. Csuri, C. (1970) "Real-Time Film Animation", Proc. 9th UAIDE Annual Meeting, pp. 289-305.

46. Csuri, C. (1974) "Real-Time Computer Animation", Proc. IFIP Congress '74, North-Holland, pp. 707-711.

47. Csuri, C. (1974) "Computer Graphics and Art", Proc. IEEE 62(4), pp. 503-515.

48. Csuri, C. (1975) "Computer Animation", Proc. SIGGRAPH'75, pp. 92-101.

49. Csuri, C., Hackathorn, R., Parent, R., Carlson, W. and Howard, M. (1979) "Towards an Interactive High Visual Complexity Animation System", Proc. SIGGRAPH'79, Computer Graphics, Vol. 13, No 2, pp. 289-299.

50. Csuri, C. and Shaffer, J. (1968) "Art, Computers and Mathematics", Proc. Fall Joint Computer Conf., AFIPS, pp. 1293-1298.

D

51. DeFanti, T. (1976) "The Digital Component of the Circle Graphics Habitat", Proc. National Computer Conference '76, pp. 195-203.

52. DeFanti, T. (1980) "Language Control Structures for Easy Electronic Visualization", BYTE, November Issue, pp. 90-106.

53. DeFanti, T. (1983) "Extended Memory Use in the ZGRASS Graphics System", in Computer Graphics, Theory and Applications, Springer-Verlag, Tokyo, pp. 380-3

54. Dietrich F. (1983) "A Micro Computer System for Real-Time Animation", The Artist Designer and Computer Graphics, Tutorial SIGGRAPH'83, Vol. 18, pp. 43-47.

55. Duncan, W. Jr (1982) "Computer Animation at Information International", Turorial notes on 3D Computer Animation, SIGGRAPH'82.

56. Duff, D.S. (1976) Simulation and Animation, M.Sc. Thesis, Dept. of Computer Science, University of Toronto.

57. Duff, T. (1983) "Computer Graphics in the Biggest Box Office Hit: Return of the Jedi", Proc. Computer Graphics '83, Online Conf., pp. 283-289.

F

58. Feiner, S., Salesin, D. and Banchoff, T. (1982) "Dial: A Diagrammatic Animation Language", IEEE Computer Graphics and Applications, Vol. 2, No 9, pp. 43-54.

59. Ferderber, S. (1983) "The Commercial Production Designer", Millimeter, February Issue, pp. 52-66.

60. Fetter, W.A. (1964) Computer Graphics in Communication, McGraw-Hill, New York.

61. Fetter, W.A. (1981) "Wide Angle Displays for Tactical Situations", Proc. US Army Third Computer Graphics Workshop, pp. 99-103.

62. Fetter, W.A. (1982) "A Progression of Human Figures Simulated by Computer Graphics", IEEE Computer Graphics and Applications, Vol. 2, No 9, pp. 9-13.

63. Fishkin, K.P. and Barsky, B.A. (1984) "A Family of New Algorithms for Soft Filling", Proc. SIGGRAPH '84, pp. 235-244.

64. Fishkin, K.P. and Barsky, B.A. (1984) "Algorithms for Brush Movement in Paint Systems", Proc. Graphics Interface '84, Ottawa, pp. 9-16.

65. Fortin, D., Lamy, J.F. and Thalmann, D. (1983) "A Multiple Track Animator System", Proc. SIGGRAPH/SIGART Interdisciplinary Workshop on Motion: Representation and Perception, Toronto, pp. 180-186.

66. Fox, and Waite (1982) "Computer Animation with Color Registers", BYTE, pp. 194-214.

67. Fox, D. and Waite, M. (1984) Computer Animation Primer, McGraw-Hill.

68. Friesen, D.P. (1969) "A Professional Animator looks at Computer Animation" Proceedings 8th UAIDE Annual Meeting, pp. 187-194.

69. Futrelle, R.P. (1974) "GALATEA: Interactive Graphics for the Analysis of Moving Images", Proc. Information Processing 74, North Holland, pp. 712-715.

G

70. Geschwind, D.M. (1982) "The NOVA Opering : A Case Study in Digital Computer Animation", Proc. Computer Graphics '82, Online Conf. pp. 325-335.

71. Goldstein, R.A. (1971) "A System for Computer Animation of 3-D Objects", Proceedings 10th UAIDE Annual Meeting, pp. 3-128 to 3-139.

72. Goldstein, R.A. and Nagel, R. (1971) "3-D Visual Simulation", Simulation, pp. 25-31.

73. Goss, T. (1983) "Animation and the New Machine", Print, March/April Issue, pp. 57-64.

74. Green, M. (1981) "A System for Designing and Animating Objects with Curved Surfaces", Proc. Canadian Man-Computer Communications Society '81, pp. 377-384.

75. Greenberg, J.M. (1985) "Computer Animation in Distance Teaching", Proc. Graphics Interface '85, Montreal.

H

76. Hackathorn, R. (1977) "ANIMA II:A 3-D Color Animation System", Proc. SIGGRAPH'77, Computer Graphics, Vol. 11, No 2, pp. 54-64.

77. Hackathorn, R., Parent, R., Marshall, B. and Howard, M. (1981) "An Interactive Microcomputer Based 3-D Animation System", Proc. Canadian Man-Computer Communications Society Conference '81, pp. 181-191.

78. Haflinger, D.J., and Ressler, P.C. (1971) "Animation with IGS", Proceedings 10th UAIDE Annual Meeting, pp. 3-227 to 3-234.

79. Halas, J. (editor) (1974) Computer Animation, Hastings House, New York.

80. Hayward, S. (1984) Computers for Animation, Focal Press.

81. Herbison-Evans, D. (1978) "NUDES 2: A Numeric Utility Displaying Ellipsoid Solids, Version 2", Proc. SIGGRAPH'78, Computer Graphics, Vol. 12, No 3, pp. 354-356.

82. Herbison-Evans, D. (1982) "Real-Time Animation of Human Figure Drawings with Hidden-lines Omitted", IEEE Computer Graphics and Applications, Vol. 2, No 9, pp. 27-33.

83. Herbison-Evans, D. (1983) "Manipulating Ellipsoids in Animation", Computer Graphics World, No. 7, pp. 78-82.

84. Herbison-Evans, D. (1983) "Hidden Arcs of Interpenetrating and Obscuring Ellipsoids", The Australian Journal, Vol. 15, No 2, pp. 65-68.

85. Honey, F.J. (1971) "Artist Oriented Computer Animation", Journal of Society of Motion Picture and Television Engineers, Vol. 80, No 3, p. 154.

86. Honey, F.J. (1971) "Computer Animated Episodes by Single Axis Rotations", Proc. 10th UAIDE Annual Meeting, pp. 3-120 to 3-226.

87. Hopgood, F.R.A. (1969) "GROATS: A graphic output system for atlas using the 4020". Proceedings 9th UAIDE Annual Meeting, pp. 401-410.

88. Hubschman, H. and Zucker, S.W. (1982) "Frame-to-Frame Coherence and the Hidden Surface Computation: Constraints for a Convex World", ACM Trans. on Graphics, Vol. 1, No 2, pp. 129-162.

89. Huggins, W.H., and Entwisle, D.R. (1969) "Computer Animation for the Academic Community", SJCC, AFIPS Conference Proceedings, Vol. 34, p. 623.

90. Hunter, G.M. (1977) "Computer Animation Survey", Comput. and Graphics, Pergamon Press, Vol. 2, pp. 225-229.

91. Hurn, B. (1981) "Computer Animation for Industrial Training", Computer Graphics World, No 10, pp. 65-68.

I

92. Iversen, W.R. (1982) "Processor Animates 3-d Surface Images", Electronics, 1982, pp. 149-150.

K

93. Kahn, K.M. (1976) An Actor-Based Computer Animation Language, MIT AI Working Paper #120.

94. Kallis, S.A. (1971) "Computer Animation Techniques", Journal of the SMPTE, Vol. 80, No 3, pp. 145-148.

95. Kawaguchi, Y. (1981) Digital Image (in Japanese), ASCII Publishing.

96. Kitching, A. (1973) "Computer Animation-Some New ANTICS", Br. Kinematography Sound Television J., 55(12), pp. 372-386.

97. Knowlton, K.C. (1964) "A Computer Technique for Producing Animated Movies", Proc. SJCC AFIPS Conference, W. 25, pp. 67-87.

98. Knowlton, K.C. (1965) "Computer-Produced Movies", Science, 150, pp. 1116-1120.

99. Knowlton, K. (1968) "Computer-Animated Movies", Emerging Concepts in Computer Graphics, Benjamin, N.Y., pp. 243-370.

100. Knowlton, K.C. (1970) "EXPLOR-A Generator of Images", Proc. 9th UAIDE Annual Meeting, pp. 543-583.

101. Knowlton, K. (1972) "Collaborations with Artists: A Programmer's Reflections", Proc. IFIP Working Conf. on Graphic Languages, North-Holland, pp. 399-418.

102. Knowlton, K.C. (1981) "Computer Animation as an Aid to Comprehending the Universe", Computers for Imagemaking, D.R. Clark, ed., Permagon Press, Oxford and New York, 1981, pp. 131-139.

103. Kochanek, D.H.U. and Bartels, R.H. (1984) "Interpolating Splines with Local Tension, Continuity and Bias Control", Proc. SIGGRAPH '84, pp. 33-41.

104. Korein, J. and Badler, N. (1983) "Temporal Anti-Aliasing in Computer Generated Animation", Proc. SIGGRAPH '83, Computer Graphics, Vol. 17, No 3, pp. 377-388.

L

105. Lansdown, R.J. (1982) "Computer Aided Animation: A Concise Review", Proc. Computer Graphics '82, Online Conf., pp. 279-290.

106. Lansdown, J. (1983) "The Economics of Computer-Aided Animation", Proc. Computer Graphics '83, Online Conf., pp. 267-275.

107. Laybourne, K. (1979) The Animation Book, Crown.

108. Levine, S.R. (1975) "Computer Animation at Lawrence Livermore Laboratory", Proc. SIGGRAPH '75, pp. 81-84.

109. Levitan, E.L. (1977) Electronic Imaging Techniques, Van Nostrand.

110. Levoy, M. (1977) "A Color Animation System Based on the Multiplane Technique", Proc. SIGGRAPH '77, Computer Graphics, Vol. 11, No 2, pp. 65-71.

111. Levoy, M. (1978) Computer-Assisted Cartoon Animation, master's thesis, Cornell University, Ithaca, N.Y., Aug. 1978.

112. Lewell, J. (1981) "The Computer Paintings of David Em", Business Screen, October Issue, pp. 38-40.

113. Lewell, J. (1983) "The Pioneers: John Whitney Sr", Computer Pictures, Vol. 1, No 3, pp. 22-24.

114. Lewis, J.P. (1984) "Texture Synthesis for Digital Painting", Proc. SIGGRAPH '84, pp. 245-252.

115. Lieberman, L.I. (1971) "Compufilms: A Computer Animation Process", Simulation 16(1), pp. 33-36.

116. Lipscomb, J.S. (1981) "Reversed Apparent Movement and Erratic Motion with Many Refreshes per Update", Computer Graphics, Vol. 14, No 4, pp. 113-118.

M

117. Magnenat-Thalmann, N. and Thalmann, D. (1983) "3D Computer Animation Films Using a Programming Language and Interactive Systems", Proc. Computer Graphics '83, Online Conf. pp. 247-257.

118. Magnenat-Thalmann, N. and Thalmann, D. (1983) "The Use of 3D High-Level Graphical Types in the MIRA Animation System", IEEE Computer Graphics and Applications, Vol. 3, No 9, pp. 9-16.

119. Magnenat-Thalmann, N. and Thalmann, D. (1983) "The Use of 3D Abstract Graphical Types in Computer Graphics and Animation", in Computer Graphics, Theory and Applications (T.L. Kunii, ed.), Springer-Verlag, Tokyo, pp. 360-373.

120. Magnenat-Thalmann, N. and Thalmann, D. (1984) "CINEMIRA: A 3D Computer Animation Language Based on Actor and Camera Data Types", Technical Report, University of Montreal.

121. Magnenat-Thalmann, N. and Thalmann, D. (1984) "3D Shaded Director-Oriented Computer Animation", Proc. Graphics Interface '84, Ottawa.

122. Magnenat-Thalmann, N. and Thalmann, D. (1985) Computer Animation: Theory and Practice, Springer-Verlag, Tokyo, 1985

123. Magnenat-Thalmann, N., Thalmann, D. and Fortin, M. (1985) "MIRANIM: An Extensible Director-Oriented System for the Animation of Realistic Images", IEEE Computer Graphics and Applications, March Issue.

124. Magnenat-Thalmann, N., Thalmann, D., Fortin, M. and Langlois, L. (1985) "MIRA-SHADING: A Language for the Synthesis and the Animation of Realistic Images", Frontiers in Computer Graphics, Springer-Verlag, pp. 101-113.

125. Malowany, A.S. and Kashef, B. (1984) "A Color Real-Time Animation System", Proc. Graphics Interface '84, Ottawa, pp. 43-50.

126. Marion, A., Fleischer, K. and Vickers, M. (1984) "Toward Expressive Animation for Interactive Characters", Proc. Graphics Interface '84, Ottawa, pp. 17-20.

127. Max, N.L. (1979) "Atom LLL: - Atoms with Shading and Highlights", Computer Graphics, Vol. 13, No 2, pp. 165-173.

128. Max, N. and Blunden, J. (1980) "Optical Printing in Computer Animation", Proc. SIGGRAPH '80, Computer Graphics, Vol. 14, No 3, pp. 171-177.

129. McCarthy, M. (1982) "Animation's New Protege", Video Systems, pp. 40-46.

130. Mezei, L. and Zivian, A. (1971) "ARTA; An Interactive Animation System", Proc. Information Processing 71, North-Holland, pp. 429-434.

131. Mittelman, P. (1983) "Computer Graphics at MAGI", Proc. Computer Graphics '83, Online Conf., pp. 291-301.

132. Miura, T., Iwata, J., and Tsuda, J. (1967) "An Application of Hybrid Curve Generation - Cartoon Animation by Electronic Computers", Proc. Spring Joint Computer Conference, p.141.

133. Mudur, S.P., and Singh, J.H. (1978) "A Notation for Computer Animation", IEEE Trans. on Systems, Man and Cybernetics, Vol. SMC-8, No 4, pp. 308-311.

N

134. Negroponte, N. and Pangaro, P. (1976) "Experiments with Computer Animation", Computer Graphics, Vol. 10, No 2, pp. 40-44.

135. Nolan, J., and Yarbrough, L. (1968) "An on-line Computer Drawing and Animation System", Proceedings IFIPS Congress 1968, North-Holland, Amsterdam, p. 605.

136. Noll, A.M. (1965) "Stereographic Projections by Digital Computers", Computers and Automation, Vol. 14, pp. 32-34.

137. Noll, A.M. (1965) "Computer Generated Three-Dimensional Movies", Computers and Automation, November Issue, p. 20.

138. Noll, A.M. (1967) "Computers and the Visual Arts", Design and Planning, 2, pp. 65-80.

139. Noma, T. and Kunii, T.L. (1985) "A Framework for Generating 3D Engineering Animation from 2D", Proc. Graphics Interface '85, Montreal.

O

140. Odgers, C.R. (1982) "Criteria for Choosing a Camera for Use in a Video Digitizing System", Tutorial notes on Computer Animation, SIGGRAPH '82, pp. 108-119.

141. Odgers, C.R. (1983) "Fundamentals of Video Recording for Computer Animation", Tutorial notes on Computer Animation, SIGGRAPH '83, pp. 175-186.

142. O'Rourke, J. and Badler, N.I. (1979) Decomposition of Three-dimensional Objects into Spheres", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. RAMI-1, pp. 295-306.

P

143. Parke, F.I. (1972) "Animation of Faces", Proceedings ACM Annual Conference, Vol. 1.

144. Parke, Frederic I. (1974) A Parametric Model for Human Faces, Ph.D. thesis, University of Utah, Salt Lake City, Utah.

145. Parke, F.I. (1975) "A Model for Human Faces that Allows Speech Synchronized Animation", Computers and Graphics, Pergamon Press, Vol. 1, No 1, pp. 1-4.

146. Parke, Frederic I. (1979) "Computer Graphic Models for the Human Face", Proc. Compsac 79, IEEE Computer Society, Los Alamitos, Calif., pp. 724-727.

147. Parke, F.I. (1980) "Adaptation of Scan and Slit-Scan Techniques to Computer Animation", Proc. SIGGRAPH '80, Computer Graphics, Vol. 14, No 3, pp. 178-181.

148. Parke, F.I. (1982) "Parameterized Models for Facial Animation", IEEE Computer Graphics and Applications, Vol. 2, No 9, pp. 61-68.

149. Patterson, R. (1982) "The Making of Tron", American Cinematographer, Vol. 63, No 8.

150. Platt, S. and Badler, N. (1981) "Animating Facial Expressions", Proc. SIGGRAPH '81, Computer Graphics, Vol. 15, No 3, pp. 245-252.

151. Porter, T. and Duff, T. (1984) "Compositing Digital Images", Proc. SIGGRAPH '84, pp. 253-259.

152. Porter, T.K. (1984) "Computer Graphics and Major Motion Pictures", Proc. CAMP '84, Berlin.

153. Potel, M.J. (1977) "Real-Time Playback in Animation Systems", Proc. SIGGRAPH '77, Computer Graphics, Vol. 11, No 2, pp. 72-77.

154. Potmesil, M. and Chakravarty, I. (1983) "Modeling Motion Blur in Computer-Generated Images", Proc. SIGGRAPH '83, Computer Graphics, Vol. 17, No 3, pp. 389-399.

155. Potter, T.E. and Willmert, K.D. (1975) "Three-Dimensional Human Display Model", Computer Graphics, Vol. 9, No 1, pp. 102-110.

156. Potts, J. (1983) "Animating the Indescribable", Government Data Systems, May/June Issue, pp. 10-13.

R

157. Reeves, W.T. (1980) Quantitative Representations of Complex Dynamic Shape for Motion Analysis, Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto.

158. Reeves, W.T. (1981) "Inbetweening for Computer Animation Utilizing Moving Point Constraints", Proc. SIGGRAPH '81, ACM, pp. 263-269.

159. Reeves, W.T. (1983) "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects", Proc. SIGGRAPH '83, Computer Graphics, Vol. 17, No 3, pp. 359-376.

160. Ressler, S.P. (1982) An Object Editor for a Real Time Animation Processor, Proc. of Graphics Interface '82, pp. 221-226.

161. Reynolds, C.W. (1978) Computer Animation in the World of Actors and Scripts, SM Thesis, MIT (Architecture Machine Group).

162. Reynolds, C.W. (1982) "Computer Animation with Scripts and Actors", Proc. SIGGRAPH '82, Computer Graphics, Vol. 16, No 3, pp. 289-296.

163. Russett, R. and Starr C. (1976) Experimental Animation, New York: Van Nostrand Reinhold.

S

164. Schachter, B.J. (1981) "Computer Image Generation for Flight Simulation", IEEE Computer Graphics and Applications, Vol. 1, No 4, pp. 29-68.

165. Schachter, B.J. (1983) "Generation of Special Effects", in Computer Image Generation, John Wiley, New York, pp. 155-172.

166. Schnéegans, C. and Poulard, S. (1972), "FILOMENE_FILEMON, Un système complexe pour la production de films d'animation sur ordinateurs", Proc. CIPS '72, pp. 212301-212329.

167. Schumacker, R.A., Brand, B., Gilliland, M. and Sharp, W. (1969) Study for Applying Computer-generated Images to Visual Simulation, AFHRL-TR-69-14, U.S. Air Force Human Resources Lab.

168. Shoup, R.G. (1979) "Colour Table Animation", Proc. SIGGRAPH '79, Computer Graphics, Vol. 13, No 2, pp. 8-13.

169. Shoup, R.G. (1979) "SUPERPAINT: The Digital Animator", Datamation, May Issue, pp. 150-156.

170. Sinden, F.W. (1967) "Synthetic Cinematography", Perspective 7, No 4, pp. 279-289.

171. Smith, A.R. (1979) "Tint Fill", Proc. SIGGRAPH '79, Computer Graphics, Vol. 13, No 2, pp. 276-283.

172. Smith, A.R. (1978) PAINT, Technical memo no 7, NYIT.

173. Smith, A.R. (1983) Digital Filmmaking, Abacus, Springer-Verlag, Vol. 1, No 1, pp. 28-45.

174. Smoliar, S.W., and Tracton, W. (1978) "A Lexical Analysis of Labanotation with an Associated Data Structure", Proc. ACM Annual Conf., Vol. 2, pp. 727-730.

175. Smoliar, S.W., and Weber, L. (1977) "Using the Computer for a Semantic Representation of Labanotation", Computing and the Humanities, Unvi. Waterloo Press, pp. 253-261.

176. Sorensen, P. (1982) "Tronic Imagery", Cinefex 8, April issue.

177. Sorensen, P. (1983) "Movies, Computers and the Future", American Cinematographer, January issue.

178. Spina, L. (1982) "Paint-By-Pixels: Computer Power Comes to TV Artists", Millimeter Magazine, Vol. 10, No 2.

179. Stern, G. (1978) Garland's Animation System - A Computer Aided System for Animated Motion Pictures, Ph.D. Thesis, University of Utah, Salt Lake City, Utah.

180. Stern, G. (1979) "Softcel: An Application of Raster Scan Graphics to Conventional Cel Animation", Proc. SIGGRAPH '79, Computer Graphics, Vol. 13, No 3, pp. 284-288.

181. Stern, G. (1983) "Bboop: A System for 3D Key Frame Figure Animation", SIGGRAPH '83 tutorial, pp. 240-243.

182. Sturman, D. (1984) "Interactive Key Frame Animation of 3D Articulated Models", Proc. Graphics Interface '84, Ottawa, pp. 35-40.

T

183. Talbot, P.A., Carr III, J.W., Coulter Jr., R.C. and Hwang, R.C. (1971) "Animator: An On-Line Two-dimensional Film Animation System", Communications of the ACM, Vol. 14, No 4, pp. 251-259.

184. Taylor, R. (1983) "Designing for the Feature Film", Tutorial notes on the Artist/Designer and Computer Graphics, SIGGRAPH '83, p. 31.

185. Thalmann, D. and Magnenat-Thalmann, N. (1983) "Actor and Camera Data Types in Computer Animation", Proc. Graphics Interface '83, pp. 203-210.

186. Thalmann, D. and Magnenat-Thalmann, N. (1984) "Towards an Artist-Oriented Approach to 3D-Computer Animation", Proc. CAMP '84, Berlin, pp. 523-527.

187. Thalmann, D. and Magnenat-Thalmann, N. (1985) "3D Computer Animation:More an Evolution Problem than a Motion Problem", Proc. Graphics Interface '85, Montreal, 1985

188. Thalmann, D., Magnenat-Thalmann, N. and Bergeron, P. (1982) "Dream Flight: A Fictional Film Produced by 3D Computer Animation", Proc. Computer Graphics '82, Online Conf. Ltd., pp. 353-368.

189. Thornton, R. (1983) "Computer assisted Animation at NYIT", Proc. Computer Graphics '83, Online Conf., pp. 277-282.

190. Tilson, M.D. (1976) Editing Computer Animated Film, master's thesis, University of Toronto, Canada.

W

191. Wallace, B.A. (1981) "Merging and Transformation of Raster Images for Cartoon Animation", Proc. SIGGRAPH '81, Computer Graphics, Vol. 15, No 3, pp. 253-262.

192. Weber, L., Smoliar, S.W. and Badler, N.I. (1978) "An Architecture for the Simulation of Human Movement", Proc. ACM National Conference, pp. 737-745.

193. Wein, M. and Burtnyk, N. (1972) "A Computer Facility for Film Animation and Music", Proc. CIPS '72, pp. 212201-212205.

194. Wein, M. and Burtnyk, N. (1976) "Computer Animation" in: Encyclopedia of Computer Science and Technology, Vol. 5, Marcel Dekker, pp. 397-436.

195. Weiner, D.D. and Anderson, S.E. (1968), "A Computer Animation Movie Language for Educational Motion Pictures", Proc. FJCC, p. 1318.

196. Whitney, J.H. (1971) "A Computer art for the Video Picture Wall", Proceedings IFIPS Congress 1971, North-Holland, Amsterdam, pp. 1382-1386.

197. Whitney, J.H. (1971) "Analog and Digital Computer Graphic Systems Applied to a New Motion-Picture Fine Art", J. Soc. Motion Picture and Television Eng., Vol. 80, No 3, p. 196.

198. Whitney, J. (1980) Digital Harmony, Peterborough, NH: Byte Books, McGraw-Hill.

199. Wilhelms, J. and Barsky, B.A. (1985) "Using Dynamics Analysis for the Animation of Articulated Bodies as Humans and Robots", Proc. Graphics Interface '85, Montreal.

200. Williams, L. (1983) "Overview of 3D Animation", Tutorial notes on Computer Animation SIGGRAPH '83, pp. 212-219.

201. Willmert, K.D. (1978) "Graphic Display of Human Motion", Proc. ACM '78 Conf., pp. 715-719.

202. Withrow, C. (1970) A Dynamic Model for Computer-aided Choreography, Univ. Utah, Dep. Computer Science, No 70-103.

Z

203. Zajac, E.E. (1965) "Computer Animation: A new Scientific and Educational Tool", J. SMPTE 74, pp. 1006-1008.

204. Zajac, E.E. (1966) "Film Animation by Computer", New Scientist 29, pp. 346-349.

205. Zeltzer, D. and Csuri, C. (1981) "Goal-Directed Movement Simulation", Proc. Canadian Man-Computer Communications Society '81, pp. 271-280.

206. Zeltzer, D. (1982) "Motor Control Techniques for Figure Animation", IEEE Computer Graphics and Applications, Vol. 2, No 9, pp. 53-59.

207. Zeltzer, D. (1982) "Representation of Complex Animated Figures", Proc. Graphics Interface '82, pp. 205-211.

208. Zeltzer, D. (1983) "Knowledge-based Animation", Proc. SIGGRAPH/SIGART Workshop on Motion, pp. 187-192.

209. Zeltzer, D. (1985) "Towards an Integrated View of 3D Computer Animation", Proc. Graphics Interface '85, Montrea.

210. Zimmerlin, T., Stanley, J. and Stone, W. (1978) "A Sensor Simulation And Animation System", Proc. SIGGRAPH '78, Computer Graphics, Vol. 12, No 3, pp. 105-110.

early papers

2 14 23 26 29 35 36 42 45 50 60 68 71 72 78 79 85 86 89 94 96 97 98 99 100 101 115 130 135 136 137 138 183 196 197 203 204

books and surveys

20 25 44 67 79 80 90 95 105 109 122 163 165 173 194 198 200

theoretical papers

4 6 133 157 187 199 209

wide-audience papers

3 5 18 19 22 30 38 40 43 44 59 66 73 91 92 106 112 113 129 140 141 149 156 170 173 176 177 178 184

key-frame animation

13 14 15 16 26 27 29 37 65 96 103 110 111 126 132 158 179 180 181 182 191

paint systems

39 63 64 96 110 111 114 169 171 172

real-time animation

1 21 45 46 48 51 52 53 66 125 153 160 168

computer animation languages

2 23 58 93 97 98 99 100 101 102 117 118 119 120 121 123 124 133 161 162 185 188 195

computer animation systems

17 24 27 28 36 41 42 48 49 51 52 53 54 65 69 71 74 76 77 81 85 86 87 96 110 117 121 123 124 125 127 130 135 166 179 180 181 183 186 190 193 197 210