INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Using behavioral simulation to animate complex processes

Haumann, David Roger, Ph.D.
The Ohio State University, 1989
USING BEHAVIORAL SIMULATION TO ANIMATE COMPLEX PROCESSES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

David Roger Haumann

*****

The Ohio State University

1989

Dissertation Committee
Dr. Richard Parent
Professor Charles Csuri
Dr. Wayne Carlson

Approved by

Adviser
Computer and Information Science
To My Wife, Family and Friends
ACKNOWLEDGEMENTS

I would like to thank Professor Charles ("Chuck") Csuri for his unfailing encouragement and support for this research project and for his vision and determination in nurturing computer animation into a mature art form. Also thanks to Dr. Thomas Linehan for making the program here exciting enough to attract such a talented cadre of artists and for providing a much needed daily dose of good humor. My fellow graduate students helped to provide an intellectually stimulating atmosphere for computer animation research, especially Michael Girard, Susan Amkraut, Brian Guenter, John Chadwick, Craig Caldwell, John Fujii, Mark Jansen, and James Hahn. Special thanks are due to David Zeltzer, who first inspired me to come to OSU, and to Chris Wedge who willingly followed me here and lead John Donkin and myself in the collaborative creation of Balloon Guy. Additional artistic inspirations and support for this research were provided by Gregory Foss, Joan Stavely, and Craig Caldwell, who along with the rest of the ultimate frisbee players was also instrumental in helping me stay physically fit.

I am grateful to my advisor Dr. Richard ("Rick") Parent for his constant support, faith, and confidence in my abilities as well as his efforts to maintain a calm and relaxed attitude towards each of the obstacles that fell in our path. I
would also like to thank my minor advisor Dr. Balakrishnan ("Chandra") Chandrasekaran for many insightful discussions on graphics and AI as well as my third reader, Dr. Wayne Carlson who deserves much credit for making this work take on a readable and coherent form.

An uncountable number of "thank you's" are due to the support staff that eagerly answered questions and attended to the myriad of details encountered in our daily operations. Without them the success of this research would have been severely jeopardized. Special thanks are due to Steve Spencer for helping format this document in \LaTeX, and to Scott Dyer and the rest of the Supercomputer staff for the countless times they have provided software advice and support. In the same way, Killer and Frog were most helpful in the earlier days back at the Cranston Center. I would also like to thank Kris Conrad, Naomi Josephson, Allison Hoffman, and Stacey Diehl, for managing all the office operations so well.

Last and most of all, I would like to thank my wife, whose loving support and self sacrifice have been the most precious gifts endowed to this undertaking. Without them this task would not have been possible.

Finally, to all those individuals whom I have failed to mention by name, I would like to extend my humble apology, but most importantly, my sincerest thanks.
Vita


1977 . . . . . . . . . . . . . . . B.S., Brown University, Providence, Rhode Island.

1985 . . . . . . . . . . . . . . . M.S., The Ohio State University, Columbus, Ohio.

Publications


Fields of Study

Major Field: Computer and Information Science
Minor Field: Artificial Intelligence, Dr. B. Chandrasekaran
# Table of Contents

DEDICATION ................................................................. ii  
ACKNOWLEDGEMENTS ................................................. iii  
VITA ............................................................................... v  
LIST OF FIGURES ........................................................... x  

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
</tbody>
</table>
| 1.1 Traditional Approaches to Motion Control ................. 4  
| 1.2 Motion Synthesis Through Simulation ....................... 8  
| 1.2.1 Advantages of Simulation ................................ 10  
| 1.2.2 Past Use of Simulation ................................... 11  
| 1.3 Behavioral Simulation ....................................... 15  |
| II      |      |
| Reductionism in Science ........................................... 19  
| 2.1 Reductionism in Physical Models .............................. 20  
| 2.1.1 Chaos ........................................................... 21  
| 2.1.2 Particle Systems ............................................ 22  
| 2.1.3 The Finite Element Method ................................ 24  
| 2.2 Reductionism in Cognitive Models ............................ 25  
| 2.2.1 Algorithmic Approach ...................................... 26  
| 2.2.2 Connectionist Approach .................................... 27  
| 2.3 Reductionism in Computer Science ............................ 29  
| 2.3.1 Turing Machines ............................................. 29  
| 2.3.2 Cellular Automata .......................................... 30  
| 2.3.3 Actor-Based Systems ....................................... 35  
| 2.4 Effects of Reductionism on Animation ...................... 50  
| 2.4.1 Actors, AI, and Animation ................................ 50  
| 2.4.2 Ani and the Director Language ............................ 53  
| 2.4.3 Murtagh ......................................................... 56  
| 2.4.4 Reynolds ....................................................... 63  
| 2.4.5 The Thalmanns ............................................... 66  |

vi
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.6 Physically-Based Modeling</td>
<td>68</td>
</tr>
<tr>
<td>2.4.7 Flexible Objects</td>
<td>71</td>
</tr>
<tr>
<td>III Behavioral Simulation in Animation</td>
<td>77</td>
</tr>
<tr>
<td>3.1 Motivation and Goals Behind Behavioral Simulation</td>
<td>77</td>
</tr>
<tr>
<td>3.2 Behavioral Simulation Defined</td>
<td>79</td>
</tr>
<tr>
<td>3.3 Comparison of Behavioral Simulation with Other Approaches</td>
<td>85</td>
</tr>
<tr>
<td>3.3.1 Behavioral Simulation and Behavioral Animation of Reynold's</td>
<td>85</td>
</tr>
<tr>
<td>3.3.2 How Cellular Automata Compares with Behavioral Simulation</td>
<td>86</td>
</tr>
<tr>
<td>3.3.3 How Particle Systems Compare with Behavioral Simulation</td>
<td>88</td>
</tr>
<tr>
<td>3.4 Characteristics and Ramifications of Behavioral Simulation</td>
<td>89</td>
</tr>
<tr>
<td>IV Test-bed for Behavioral Simulations</td>
<td>93</td>
</tr>
<tr>
<td>4.1 Test-bed Design and Implementation Overview</td>
<td>93</td>
</tr>
<tr>
<td>4.2 The Advantages of an Object-Oriented Approach</td>
<td>98</td>
</tr>
<tr>
<td>4.3 The Environmental Hierarchy</td>
<td>99</td>
</tr>
<tr>
<td>4.4 State History</td>
<td>105</td>
</tr>
<tr>
<td>4.5 Scripting the Properties of Elements</td>
<td>107</td>
</tr>
<tr>
<td>4.6 Stepping Along in Time</td>
<td>108</td>
</tr>
<tr>
<td>4.7 Visual Representation</td>
<td>113</td>
</tr>
<tr>
<td>4.8 How the Test-bed was Implemented</td>
<td>114</td>
</tr>
<tr>
<td>4.9 How the Test-bed is Used</td>
<td>114</td>
</tr>
<tr>
<td>V Experimental Results</td>
<td>117</td>
</tr>
<tr>
<td>5.1 Goal Directed Motion</td>
<td>118</td>
</tr>
<tr>
<td>5.2 Physically Based Model for Flexible Objects</td>
<td>119</td>
</tr>
<tr>
<td>5.2.1 The Mass Actor</td>
<td>120</td>
</tr>
<tr>
<td>5.2.2 Environmental Forces</td>
<td>121</td>
</tr>
<tr>
<td>5.2.3 The Spring Actor</td>
<td>121</td>
</tr>
<tr>
<td>5.2.4 The Hinge Actor</td>
<td>122</td>
</tr>
<tr>
<td>5.2.5 The Aerodynamic Drag and Wind Actors</td>
<td>122</td>
</tr>
<tr>
<td>5.2.6 The Gravity and Ground Actors</td>
<td>123</td>
</tr>
<tr>
<td>5.2.7 The Contact Actor</td>
<td>123</td>
</tr>
<tr>
<td>5.2.8 The Body Actor</td>
<td>124</td>
</tr>
<tr>
<td>5.2.9 Animated Results</td>
<td>125</td>
</tr>
<tr>
<td>5.3 Artificial Life - a Moth</td>
<td>140</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Sample cellular automata transition function .......................................... 32
2.2 Sample cellular automata evolution ........................................................... 32

3.1 Network of relations in Times Square ....................................................... 83

4.1 System components ................................................................................ 96
4.2 Interaction window ................................................................................. 97
4.3 Environmental hierarchy node implementation ....................................... 100
4.4 Example environmental hierarchy ............................................................. 102
4.5 Vector script example ............................................................................. 109
4.6 Transformable script .............................................................................. 110

5.1 Non-deformed falling sphere ................................................................. 128
5.2 Deformed sphere “squashing” on floor .................................................... 128
5.3 Sphere rebounding .................................................................................. 129
5.4 Non-deformed falling rod ...................................................................... 129
5.5 Deformed rod striking the floor ............................................................... 129
5.6 Rod rebounding and rotating ......................................................... 130
5.7 Rod (far end now striking floor) ................................................. 130
5.8 Stroboscopic path of falling paper .............................................. 132
5.9 Stroboscopic closeup of paper “flipping” .................................. 132
5.10 Paper beginning to slide “downhill” ......................................... 133
5.11 Paper falling “downhill” ............................................................. 133
5.12 Paper beginning to move “uphill” .............................................. 133
5.13 Flag with ripple entering from left .......................................... 135
5.14 Flag with ripple midway through ........................................... 135
5.15 Flag with ripple moving off right side ..................................... 135
5.16 Flag released from flagpole ...................................................... 137
5.17 Static curtain shape ................................................................. 137
5.18 Curtain catching the breeze ..................................................... 137
5.19 Curtain settling back ............................................................... 138
5.20 Characters from the film Balloon Guy .................................... 138
CHAPTER I

Introduction

The virtuosity of an artistic medium will invariably benefit from improved capacities to convey visual complexity. Visual complexity in computer graphics is the available "bandwidth" in which the informative content of a computer generated scene may be carried. Informative content is introduced by ordering and controlling the complexity in a manner which is comprehensible to the viewer. The ability to generate and control visual complexity is a prerequisite for increasing the informative content and hence the interest and appeal of artistic computer graphic creations.

The field of computer graphics has grown quite sophisticated in terms of the complexity and informative content conveyable in a computer generated scene. Current techniques are now capable of producing images of stunning realism and of sufficient complexity to capture and hold the attention of most viewer-patrons of this new found art form. For example, ray-tracing techniques introduced by Whitted [Whitted 1980] accurately reproduce the illusions of reflected and refracted
light passing through transparent media in a simulated world. Radiosity techniques [Cohen 1985][Immel 1986] capture the subtle diffusive qualities of interior scenes lit by artificial light. Fractal techniques [Mandelbrot 1983] can be used to mimic the complexity found in a wide variety of natural scenes, especially plants and mountains. Texture mapping and bump mapping [Heckbert 1986] allow the surface colorations and shadings of almost any material to be convincingly recreated in a computer generated scene. Each of these techniques enhances both the complexity as well as the informative content necessary to synthesize realistic and hence believable imagery.

Computer animation is computer graphics with the additional dimension of time. As an extension of computer graphics, each of the techniques listed above adds to the informative complexity of animated scenes. While the dimension of time provides us with more bandwidth for visual communication, it challenges us to find ways to generate temporal complexity and to mold this complexity into meaningful information. Our virtuosity in the computer animation medium depends in part on our ability to generate and control temporal complexity so that we may better create believable, realistic motion in complex scenes.

Until recently, this challenge has been largely unmet. For most commercially produced computer animations, static, motionless backgrounds are the norm. While this is sufficient for simulating indoor scenery, it is often insufficient for capturing the complex motion often found outdoors in nature. For commercially
produced scenes which do exhibit temporal complexity, the motions are often limited to monolithic bodies passively floating along splined paths, or slightly more complex articulated figures. Only recently have scenes with flexible and fluid qualities been created. These new advances must be pushed farther if we are to capture the complexity inherent in the flow of water, billowing smoke, or the hair and clothing on an animated character.

The main concern of this dissertation is to explore how informative temporal complexity may be generated and controlled in computer animation. The most frequent use of the temporal element in computer animation has been to depict the motion of three dimensional objects. For this reason, this research has focused on motion control problems in three dimensional animation as they relate to the generation and control of temporal elements in a scene. The approach presented here is to use a form of simulation termed "behavioral simulation" to procedurally generate motion.

The remainder of this introduction will review how traditional animation affected the development of motion control techniques in computer animation. Following this, simulation will be presented as a natural means to more fully utilize computational resources available in computer animation.
1.1 Traditional Approaches to Motion Control

The earliest models of computer animation were taken directly from techniques used in traditional animation. The keyframe technique is a process wherein full frames of animation are drawn by an experienced animator. These frames represent the entire animated state at particular "key" points in time during a story, usually the extremes of motion. These frames are handed to an assistant animator who proceeds to draw most of the major frames between the keyframes. Once most of the frames have been drawn by the assistant animator, they are handed off to the "inbetween" animator, who draws the remaining frames required to fill in the complete story. Thus, the keyframe technique, as adapted to 2D computer animation consisted of constructing full frames of animation and then filling in the inbetween frames (so called "inbetweening" [Thomas 1981]) via some type of interpolation on the drawn images.

Perhaps because the original technique of keyframing was so readily adapted to 2D computer animation, it was naturally extended to 3D computer animation as well. The 3D keyframe technique consists of specifying the position and orientation of each object in the scene for each keyframe. To generate the inbetween frames, the position and orientation values for each object are interpolated from one keyframe to the next. The interpolated values are then used to position the objects in each of the inbetween frames.
There are several drawbacks to this approach. The first is related to the fact that keyframes do not treat the different motions in a scene independently; the position and orientation for every object must be specified in each keyframe. This requires the storage of large amounts of state information, which can be redundant if some of the values could be interpolated from the previous to the following keyframe. This problem is exacerbated when numerous discontinuities are to be introduced for particular elements in the scene. A keyframe must be specified for each point in time where such discontinuities are to occur, further increasing the redundancy of the stored information.

Early on, traditional animation encountered similar problems caused by drawing independent motions in the same frame. For example, several moving elements (say two characters and a moving background) might be drawn together on the same sheet of animation bond paper. If a correction was needed in the motion of just one element, the entire frame (both characters and the background) had to be redrawn. The best solution patented by Earl Hurd in 1914 (and which has survived until this day [Crafton 1982]), was to use clear celluloid sheets ("cels") which could be placed several layers deep over a painted (possibly moving) background. A separate layer could be used for each moving element in a scene (one for each character, in this example). In this manner each motion could be animated separately.
The idea of separating the motion of objects into independently controlled elements was adapted to 3D computer animation in the form of so-called “tracks” [Gomez 1985][Fortin 1983], which are analogous to the “cels” described above. A track is an ordered series of events in time. Each event consists of a value and the time (or frame) that that value is to be attained. These events may be thought of as control points which define a function parameterized by time. The value of the function at any given time may be determined by interpolating between those events which are adjacent in time. The function a track represents may be used to control the value of one or more dynamic elements in a scene. In this way, the elements may be controlled independently. The animation is generated by applying the interpolated function values to the appropriate scene elements at each frame in the animation.

Interpolated tracks are usually used to represent spline curves parameterized by time. These curves allow the user to control the shape and continuity of a value as it changes over time. If the values are used to represent a position in space, then they are quite useful in producing smooth curved motion. Velocity along the curve can be controlled by the placement of the control points, or better, by reparameterization of the curve [Kochanek 1984].

The primary limitation of the approaches described above is that all motion must be explicitly described by the animator. While the use of interpolation techniques does remove the burden of specifying the location of each moving object
at each frame, it still requires that the animator mentally consider and manually specify the motion for every moving element in the scene. This places a severe limit on the motion complexity that can be introduced by an animator in any given animation.

One technique used in both traditional and computer animation which helped increase the motion complexity and realism in animated scenes is the technique of rotoscoping. This technique consists of filming sequences of live subjects and then either tracing (in traditional animation) or digitizing (in computer animation) the motion into the animated frames. This technique is limited however to concrete physical behavior that can be filmed. In addition, the digitization step still remains primarily a manual technique, at least until image recognition systems are perfected for this purpose.

What is needed, and what is addressed in this dissertation, is a means to automate the motion generation task for an animated scene. We wish to generate motion complexity by a means which vastly reduces the required input of the animator. In effect, we wish to apply the principle of “database amplification” (introduced by Smith in [Smith 1984]) to motion control, wherein voluminous quantities of motion may be generated from a small animator specified database.

Another limitation of the techniques described above is that the information used in the construction of the objects in a given scene is unavailable during the task of specifying the motion for those objects. Motion specification occurs in
a vacuum that lacks the notions of what the objects are, and why the animator wishes to move them as such. Although we are not proposing solutions to the knowledge representation problems addressed by the field of Artificial Intelligence (AI), we, along with other researchers ([Kahn 1979][Murtagh 1985] and [Zeltzer 1983]), do feel strongly that successful animation systems of the future will contain more information about the nature of objects to be animated, and the animator's intentions in creating the animated scene. This aspect of animation creation is important because animators use motion to convey information about the objects being moved or about the relationships between moving objects; in effect, motion helps to "tell the story". Such information about the objects, relationships, and story, if made available to the motion generation software, could be used to help lighten the motion specification burden.

1.2 Motion Synthesis Through Simulation

Pagels [Pagels 1988] writes that the computer is the new instrument in the sciences of complexity. Mathematical or logical models are being created in algorithmic form and then used to test theories about the system they are modeling. This is happening in the fields of computational astrophysics [Arnett 1985], experimental mathematics [Campbell 1985], computational biology and non-linear dynamics, to name just a few. Each of these fields is utilizing the computer as a tool to simulate a complex system in order to answer questions about the behavior of the system.
under a proposed set of circumstances. The answers to these questions are sought in the hope of constructing or verifying a proposed theory about such a system.

The research presented in this dissertation has sought to explore ways in which a similar approach might be turned around and used in the process of synthesizing particular elements used in artistic animations. Although simulation has been implicitly in use by artists when they modeled the visual attributes of their imagined realities, we are bringing in a new realm of computer simulation for motion control. Computer simulation for animated motion allows the artist to experiment with rules for constructing a system by having the computer simulation produce the required motion. By iteratively modifying the rules and initial conditions of the simulation, the achieved motion can be refined. The process ends when the motion satisfies the artistic constraints as judged by the user.

The key idea stated in the previous section is that we wish to utilize information about animated objects in order to automate the generation of their motion. It is natural to assume that the form of this information would be consistent with the way animators would conceive and mentally manipulate these objects. Such information might include notions about why particular objects are to move (causality) and what motive and temporal relationships exist between them. If we can represent natural objects which the animator wishes to animate along with their respective natural forms of motion in a useful procedural form, then the motion can be algorithmically generated. From the animator’s standpoint it becomes
convenient to view animation as a form of simulation, in which an animator is really the creator of a simulated world in which causes and effects actively move the story along to its logical, if not predictable conclusion.

1.2.1 Advantages of Simulation

Simulation is a useful and powerful means to utilize information about an object to generate its motion. First, it is a straightforward means to generate temporal complexity in computer generated motion. With the motion of objects procedurally specified, the available computational resources can be used to generate the motion of large collections of objects. High level control of the animation can be exercised by editing the properties and procedures that affect the motion of the objects. Secondly, by borrowing knowledge from engineering and scientific disciplines, convincingly realistic motion may be generated. This is important in the cases where the success of an animated story depends in part on the accurate portrayal of the physical properties of an object, properties which can only be conveyed through the quality of motion exhibited by that object. Thirdly, simulation can be used to capture the manners in which objects interact. Here again, qualities of interaction may be important for conveying information in an animated scene. One side effect of representing object interaction is that when an animator edits the motion of one object in an interacting group, the procedural generation of motion ensures that the resulting group members respond appropriately to the new changes in motion.
of the edited object. Finally, we would argue that simulation is a straightforward means in which the motion model can be described in a manner that is consistent with the user's mental model of the objects in the scene. This is important in a field where the user often lacks technical expertise.

It is important to point out that the viewpoint of simulation taken on by this research differs from the kind of simulation used in more technical endeavors. In animation we are concerned with using simulation to synthesize motion. This differs from scientific uses of simulation in order to analyze a system so as to predict its behavior, or for training purposes. Simulation for analysis has been used in a variety of areas. For example, mechanical simulation may be used to predict the performance of a physical system under certain predetermined conditions which may result in changes in design, such as using wind tunnel tests to help determine and, if necessary, correct the aerodynamic properties of aircraft designs. Computers can also be used to simulate the motion of airmasses in order to predict the weather. Finally, simulation may be used for training purposes, such as flight simulation, in which pilots may be trained at a fraction of the cost (and no risk) compared to actually flying in a real airplane.

1.2.2 Past Use of Simulation

Simulation techniques have been used to generate animated motion in several instances in the past. The particle system technique used by Reeves [Reeves 1983]
was used to capture the effect of a storm of fire traversing the surface of a planet in the motion picture feature film *Star Trek II: The Wrath of Kahn* [Paramount 1982]. Although the particle generation and life was stochastically controlled, the effects of a simulated gravity were used to cause the particles to curve in ballistic arcs as they fell to the planet's surface. Later this same technique was used to model a field of grass blowing in the wind [Reeves 1985]. Both the wind and the resulting motion of the bending grass were controlled by stochastic models. The wind was represented as a two dimensional vector field whose strength and direction at each point was determined by the density of stochastically generated "wind gust" particles. The dynamics of the bending response of a blade of grass were also varied stochastically. Because the generated scenes contained on the order of a million particles, the dynamic equations were simplified to the minimum required to achieve a desired visual effect. Nonetheless, the sheer number of randomly varied blades of grass, responding to simulated gusts of wind yields a complex and convincingly dynamic scene.

Lundin [Lundin 1984] used simulation to accurately portray the motion of mechanical robots and their physical appendages. In scenes from their film *The Works*, a robotic ant walks as its humanoid robotic driver is bounced along in his driver's seat. Upon arriving at the construction site, the flexible antennae realistically bounce as they are extended. A wheeled "driller" robot bounces, pitches, and rolls in response to the uneven terrain upon which it travels enroute to its
workplace. A robotic welder spews sparks that fall in natural arcs as they sputter to the ground.

Each of these effects was the result of using a dynamic simulation to accurately portray mechanical effects. The motions of the "driller" robot were captured by explicitly solving the force equations that described the translation and rotational dynamics of a simulated suspension system consisting of springs and shock absorbers. These equations were cast into the form of difference equations and balanced for each frame of the animation. The motions of the sparks from the welding operation were simulated using the customary ballistic equations which model the flight of objects under the influence of gravity. The initial velocities of the sparks were determined stochastically and then recomputed to simulate a bounce following a collision with an object in the scene.

Reynolds [Reynolds 1987] has used simulation to model the activities of more animate creatures. He used his technique to model flocks of birds and schools of fish when making the film *Breaking the Ice* [Symbolics 1987]. His "behavior animation" technique attempts to model the physical as well as mental behaviors of individual members (actors) of a larger group (flock or school). Rather than having the animator explicitly script the motion of each actor in advance, the motion results from the procedurally controlled behaviors of the actors as they interact with their fellow actors and the environment. The model accounts for the kinematic effects of banking and thrusting and also the perception of other
individuals and obstacles in the environment. The group behavior is achieved by simulating the collision avoiding behavior, the velocity matching behavior and the centering effects of members acting as a group.

There has been a number of researchers interested in simulating the motion of articulated figures, work which is vital if animation is to ever convincingly portray human characters. Several researchers ([Chadwick 1988] [Badler 1979] [Wilhelms 1987] [Girard 1989] [Armstrong 1987] [Zeltzer 1984]) used simulation in some form or another to model human figures. Some even advocate the use of dynamics to make the motion appear more realistic.

More recently, there has been a surge of research aimed at directly simulating the dynamics of physical objects to obtain realistic animated motion. Hahn [Hahn 1988] used classical mechanical techniques to model rigid body motion as well as the forces of friction and those resulting from collisions. Led by Alan Barr at Caltech, a number of researchers ([Terzopoulos 1987][Terzopoulos 1988a][Terzopoulos 1988b] [Barzel 1988][Platt 1988][Witkin 1988]) have used sophisticated engineering techniques to accurately model and control the motions of physical objects. This author [Haumann 1988] has also developed physical models of flexible objects. Miller [Miller 1988] has used physical models to simulate the motion of snakes and worms.
1.3 Behavioral Simulation

Many of the examples of animation cited above were specific simulation models designed to simulate one kind of phenomena. For example, Lundin’s models depended upon explicit equations that were derived directly from the geometry of the particular object to be animated. Many of the articulated figure models depend upon the geometry of the figures. While the work in physically-based modeling is the most general, it does not capture the notions of objects with self-motivated behaviors. However, the works of Reynolds and Reeves seem to have a common qualities which are more general than the particular applications in animation in which their models were used. These qualities can be summarized as follows.

The first quality is that the final animated image resulted from the use of procedural techniques to generate the motion. Reeves relied upon simple models of the life cycle of a particle to model fire or wind and bending models for the grass in the wind. Reynold’s modeling of the mental state of interacting actors generates the flock behavior. In both of these approaches, the behaviors cited in the particular examples could be enhanced, or replaced by entirely different behaviors to accommodate different models of entirely different phenomena.

A second quality, which is a consequence of the first is that the complexity of the animated scenes results from the ability to populate the scene with multiple instances of a single procedurally controlled element. In the case of Reeves work, a
million or so particles are needed to describe a scene. Reynold’s flocks consisted of considerably smaller collections (tens to hundreds) of elements. However, in this simulation, each element (bird or fish) was considerably more complex procedurally than the individual particles in Reeves’ system. The use of multiple instances of singular homogeneous (or nearly so) elements to obtain complexity is not practical unless these elements are procedurally controlled. Thus, this approach can facilitate the achievement of increased motion complexity in animated scenes.

A third quality, less evident perhaps, but of central importance to this thesis, is the global character often taken on by these large collections of elements as recognized by the viewer. By populating a scene with more elements than can be individually perceived by the viewer, “emergent” behaviors, behaviors which are not inherent within the individual elements themselves, begin to appear. Naturally, for the architects of these scenes, these characteristics are usually desirable. Again, we argue that this effect is the result of a large collection of procedurally controlled elements in a scene. The viewer perceives a flock as the result of the activity of the group as a whole, even though there is no explicit definition of the flock within the software generating the animation. The viewer perceives wind on grass because of the apparent collective motion of the clumps of grass. This effect of larger scale phenomena arising from models built from large collections of smaller elements will be termed “emergent” in the sense used by Langton [Langton 1988].
The main aim of the research contained herein has been to capitalize upon these three areas of phenomena in an attempt to generalize them and make them useful for animation. The basic idea has been to create an environment for exploring the generation of animation utilizing large collections of elements whose behaviors are procedurally defined and whose activities and interactions result in the viewer perceiving emergent phenomena that arise from the collective activity of the individual elements. We argue that models of natural phenomena which are generated in this "bottom up" fashion are more robust and hence have a better chance of "fooling the viewer" and are therefore more suited to producing convincingly realistic motion. In addition, we believe that in the absence of existing models of any such phenomena (natural or imaginatively contrived), they are more easily constructed in this bottom up fashion rather than deriving a surface model from the "top down". Finally, the general nature of this approach allows for a variety of particular examples to be exercised within a framework that can remain basically unchanged.

The approach presented here is that complexity in animation may be generated and controlled using a technique termed "behavioral simulation". Its basic premise is that interesting large scale phenomena may be simulated by the aggregate behavior of large collections of smaller elements interacting only locally. This technique achieves this effect by reducing a system to its component parts and simulating how they interact over time. The net result is that the global behavior
appears to emerge from the local interactions.

The remainder of this dissertation will attempt to present behavioral simulation in perspective with other reductionistic disciplines which have either had a direct impact upon generating and controlling complexity in computer animation, or disciplines which are related to the generation of complexity from large collections of locally interacting elements. Once this background has been established, the development of the behavioral test bed is described and the implementation of specific models of animation are examined (chapter IV). Experimental results are presented in chapter V and conclusions are drawn in chapter VI about how this technique can and cannot be useful in generating and controlling complexity in animation.
The success of the natural sciences in providing models of behavior for the world around us is due in part to its ability to abstract the relevant properties and components of a complex system and provide explanatory hypotheses about the functioning of the whole system or parts thereof in terms of these component properties and behaviors. This reductionistic view of nature, the taking apart of a complex system and *analyzing* its behavior in terms of its component parts, has been the primary methodology used to deal with complexity throughout the theoretical as well as applied sciences. It is also a powerful means with which one can *synthesize* models of a system in order to simulate its overall behavior. This idea lies at the heart of the work on behavioral simulation presented in this dissertation.

The remainder of this chapter will present examples of reductionistic approaches used in the areas of physical models, cognitive models and computer science. This presentation provides a framework from which the behavioral
simulation approach may be compared. In addition, several of these models have been directly utilized by the field of computer animation: most notably, physically-based models and actor-based systems. A final section describes such current applications to animation.

2.1 Reductionism in Physical Models

Physical models are of great importance to animation due to their applicability in rendering dynamic scenes of natural phenomena. In this section, chaos is discussed first because it sheds some light on the failures of modern day science to explain physical phenomena common to our every day experiences; for example, clouds and turbulence in running water. Because these experiences are so common, animators often expend much time and energy trying to reproduce them. In fact, the Disney animation studios had a special effects department which was devoted in part to studying physical models [Thomas 1981]. The failure of science to provide understandable, tractable models which can duplicate these effects visually has served as a source of some frustration to computer animators.

The remaining sections discuss particle systems and the finite element method; how they reduce system complexity and how the have been applied to a variety of physical systems.
2.1.1 Chaos

A new science has arisen called chaos which provides a new way of describing the behavior of complex systems (see [Gleick 1987], [Crutchfield 1986]). Traditional science has in the past labored under two viewpoints regarding systems. The first view was that simple deterministic rules governing the behavior of simple systems would result in long term stability and predictability of those systems. The second idea, on the other hand, said that complex unpredictable systems must be governed by complex rules of behavior or (in order to explain the apparent unpredictability) at least be subject to random external disturbances. These working assumptions grew from the mathematical bias towards representing (more often approximating) systems with linear equations wherever possible. It is no small wonder that linear systems do adhere to the two viewpoints expressed above. However, the vast majority of natural phenomena in complex systems can only be described by non-linear equations. Unfortunately, it is these same phenomena that we encounter each day in our lives that computer animators wish to model (recall the ocean scenes during the whale sequence in the Disney film *Pinocchio*).

Chaos shows us that a useful way in which to characterize non-linear systems is by the paths through the state space that the system travels over time. The geometric and topological characteristics of these paths reflect qualitative properties concerning the stability and predictability of the system over time. If the natural phenomena that we are trying to model is truly chaotic, it will mean two
things. First, it will imply that the simplest description of the phenomena will be a simulation of the system itself; in other words we cannot write down and use an analytic equation to compute the system state over time. Secondly, if we simulate such a chaotic system, the course of the simulation will not be controllable from the initial conditions alone since the system will exhibit extreme sensitivity to initial conditions.

2.1.2 Particle Systems

Perhaps one of the most extreme examples of reduction in the sciences is the use of single particles to represent homogeneous elements in complex systems. Particle systems are useful where the phenomena under study can ignore the individual characteristics of the elements and concentrate on their more important collective behavior. Particle systems are of interest to computer animation from both the standpoint of their ability to model non-rigid "fuzzy" objects and natural physical phenomena.

The use of particle systems in computer generated imagery was first seen by mass audiences in the movie Star Trek II: The Wrath of Kahn [Paramount 1982]. This particular use of particles was described to the graphics community by Reeves [Reeves 1983] as a means to represent and display amorphous forms. A wall of fire was represented by stochastically generated particles which followed simple laws which governed their appearance, motion, and extinction from the system.
Particle systems have long been in use in the physical sciences. They have been used in classical celestial mechanics, and in modern sciences of plasma physics, fluid dynamics and molecular dynamics. Hockney and Eastwood [Hockney 1981] provide a good survey of techniques in the scientific realm for using particle systems. One common use of particle systems is to model so called "N-body problems". These are physical situations where the activity of each body in the system depends upon the state of all other bodies in that system. For example, in a celestial system the orbits of all bodies are affected by the gravitational pull from each other (all possible pairs of interactions must be considered). When setting up a computational model of such a system the naive approach would yield an $O(N^2)$ algorithm. Under the proper conditions, however, this can be reduced to $O(N \log N)$ using a particle-mesh technique. This technique represents the combined effects of all particles as a potential field in space approximated by a discrete grid of points. At each step during a simulation, the potential field is calculated at each grid point, and then the field effect to be applied to each particle is interpolated from the grid. An approach now presented by Greengard [Greengard 1987] has determined a hierarchical means to reduce this calculation to $O(N)$ time.

Celestial Mechanics

The representation of celestial objects as points in space is a classic example of the use of particle systems. Newton's description of the solar system depended upon
reducing what is essentially an N-body problem to a set of independent equations characterizing the gravitational attraction between the sun and the nine planets. This reduction was accomplished by discarding many pairs of heavenly interaction as having inconsequential effects upon the calculated outcome.

**Fluid Flow**

Particles have been used to model fluid flow problems, especially in cases where the behavior of the individual particles must be modeled in order to capture phenomena which has so far escaped analytic description. One example of such phenomena is the transition from laminar to turbulent flow. Lattice gas models, which provide another example of reductionism in computer science, are proving useful in fluid flow research (see section 2.3.2, page 34).

### 2.1.3 The Finite Element Method

The finite element method (FEM) has its origins rooted in the theory of structures. Before the inception of FEM at the turn of this century and its later active development during the 1950's, structural analysis was performed using a "truss and framework" method in which complex structures were built up from a number of simpler elements whose characteristics were known. The final set of simultaneous differential equations for the entire structure was constructed by combining the
equations from all the elements. This set of equations described the structural unknowns as a continuous function over the region in question.

The finite element method was developed to handle cases in which analytical methods failed to provide solutions due to irregular geometries or inherent discontinuities within the problem domain. The method consists of reducing the continuous model into a set of discrete elements each with a finite number of degrees of freedom. The discrete solution is approximated at the boundaries of each element and uses polynomial approximations to determine the values of physical parameters within each element. The net effect is the approximation of a continuous function by a piecewise linear combination of algebraic polynomials.

2.2 Reductionism in Cognitive Models

The fields of artificial intelligence (AI), cognitive science and computer science in general have all been concerned with providing cognitive models of human intelligence. The goal of AI is to endow the computer with the capability of executing tasks that until now only humans could perform. For example, computers are comparatively much faster and more accurate at performing numerical computations, yet they are still incapable of intelligent behavior such as recognizing and understanding written and spoken language. In many situations the animation of a scene requires the simulation of an active, self-motivated being, an intelligent character, if you will. To this end, it may be necessary to build a model of the intentional
elements which determine the character's behavior. Naturally one would expect that such models of intelligent behavior might serve as templates for constructing intelligent computer animated characters. Once constructed, the models would drive the animation of the simulated character.

This section will describe two complementary approaches to constructing cognitive models. The first is the algorithmic approach which has been embraced by the field of AI. This approach attempts to describe and define intelligence in terms of symbolic information processing tasks that intelligent agents appear to perform. The second approach is the "connectionist" approach which takes a more neurologically oriented view of cognition. Although this approach is not new, it has recently experienced a massive resurgence of interest. Both of these approaches reduce the complexity of intelligent problem solving by structuring a collection of computing entities to seek a solution collectively. The granularity and organization properties are quite different.

2.2.1 Algorithmic Approach

The field of Artificial Intelligence is concerned with constructing theories that would explain the activity of intelligence in terms of the information processing of symbols which represent knowledge. Early ideas about representing knowledge were based upon logic. Logic was used to make logical statements about the world and then an inference engine was applied to those rules in order to make new
inferences about the world and extract new information. This approach proved useful for theorem proving. However, it has drawbacks when dealing with subtler kinds of thinking that people do in ordinary life. Such thinking usually involves incomplete knowledge and assumptions based upon current operational context.

Another means of doing this has been to represent intelligent agents as a hierarchy of agents, each child agent responsible for a subtask of the parent [Minsky 1986]. Brown [Brown 1985] has used this idea to construct expert systems based upon a hierarchy of specialists model. It has also been successfully used in a medical diagnosis system which classifies particular types of diseases based upon symptoms and the results of laboratory testing [Chandra 1983]. The hierarchy classifies the disease using an "establish-refine" mechanism. A parent node first establishes that it has enough supportive evidence to claim the disease is of a particular type, and then refines that judgement. The node which finally succeeds identifies the disease. In contrast to earlier approaches which separated the facts from the reasoning mechanism, this approach coded both kinds of information at the appropriate level in the hierarchy.

### 2.2.2 Connectionist Approach

The connectionist approach to cognition considers the physical organizational structure of the brain as a primary model that, if properly imitated, can produce intelligent behaviors. Reductionism in computer science is exemplified in the sense
of how neural networks mimic the model of the brain by describing it as a collection of simple computing elements. This idea is not new, having been inspired by the work of McCulloch and Pitts performed in 1943. These ideas reached their height of popularity amongst researchers when Rosenblatt introduced his ideas on perceptrons and but then fell into disfavor when Minsky and Papert wrote their piece, *Perceptrons* in 1969 which mathematically proved the fundamental computational limits of a single level of perceptrons. Due to this proof, connectionist research was not well funded during the 70's. Now, however, they have experienced a resurgence due to new understandings of the brain brought about by advances in the field of neuroscience as well as the development of readily available parallel computing architectures. Such architectures enable the running of experimental tests of connectionist designs to attack certain problems in cognition. Neural nets seem well suited for solving problems in perception such as vision and speech recognition for which "traditional" artificial intelligence approaches have made little progress. Such problems in perception are characterized by the apparent presence of associative memories and the capabilities for learning. Our understanding of the massive parallelism of the brain seems to support the optimistic connectionist claims in these areas. For example, the firing time of neurons is on the order of a thousand per second, yet it takes only a fraction of a second for a person to recognize a face, let's say. This is enough time for only about a hundred neuronal firings, yet
a comparable computation (if one should exist) on a serial machine might take several more orders of magnitude of operations to complete the same task.

The associative memory of connectionist models have been likened to an N-dimensional space where each dimension represents the state of a particular neuron ([Pagels 1988], pp. 129). The current state of the network is represented as a point in this N-dimensional space. The contents of the memory are represented as stable "firing" states of the entire network. There is one such stable state for each item in the memory. These stable states correspond to fixed point attractors in chaotic (dynamical systems) theory. Thus we can think of the memory contents as a surface in N-space, and the current state as a massless marble suspended in thick honey. Where ever we place the marble initially, we know it will settle into the nearest "basin" to recall that particular item in memory.

2.3 Reductionism in Computer Science

2.3.1 Turing Machines

The most salient use of reductionism in theoretical computer science is the Turing machine. As a simple model of an algorithm, the Turing machine embodies the notion of a finite stored program which can be mechanically executed in a series of discrete steps. The Turing machine serves to represent the encapsulation of the necessary characteristics of modern computers, in the abstractions of the finite state and writable input tape. It is the successful reduction of these characteristics
to the Turing machines apparent simplicity that allows them to be rigorously analyzed by strict mathematical argument.

2.3.2 Cellular Automata

"Science has little use for models that slavishly obey all of our wishes. We want models that talk back to us, models that have a mind of their own. We want to get out of our models more than we have put in. A reasonable way to start is to put in as little as possible."

([Toffoli 1987] pp. 142.)

In contrast to the Turing machine which represents a simple model of serial computation, cellular automata represent a simple model of parallel computation in which the concepts of data and computing devices are combined. Although von Neumann is popularly associated with the single serial CPU computer architecture, he was well aware of the superior processing power available in an array of processors. The notion of cellular automata was suggested to John von Neumann by Stanislaw Ulam to provide a more realistic model for the behavior of complex systems distributed in space. The ideas were useful to von Neumann in his attempts to find a reductionistic explanation of certain aspects of biology, in particular, a characterization of the complexity of information required by a system capable of self-reproduction [Kendall 1984].
A cellular automaton is an array of identical computing elements interconnected to its immediate neighbors. Each computing element maintains a current state, and all elements simultaneously undergo a transition at discrete regular time intervals. The state transition of all elements is determined by a single rule which maps the current state of each element into its future state. This mapping is a function of the current state of the element and the states of its immediately connected neighbors. The range of neighbors to which an individual element is connected is termed the "neighborhood".

For example, consider a one dimensional cellular automaton consisting of \( N \) elements where the neighborhood of the \( i \)th element consists of those elements to the immediate left \((i-1)\)th) and right \((i+1)\)th of the given element. One can define the transition function in the form of a table which defines the new state for the \( i \)th element (the current element under consideration) in terms of the old states of the left, "center" (ith), and right hand elements. For example, assume the transition function shown in figure 2.1.

A cellular automata "experiment" is run by specifying the initial state for the entire array and then computing the new state of the array from the old. Consider a cellular automata consisting of 8 elements and initialized to the state 01011110. The progression from this initial state would proceed as shown in figure 2.2.

Cellular automata are capable of displaying a rich variety of phenomena. Different rules and different starting conditions lead naturally to different patterns.
<table>
<thead>
<tr>
<th>old</th>
<th>old</th>
<th>old</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>i−1</td>
<td>i</td>
<td>i+1</td>
<td>i</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.1: Sample cellular automata transition function

\[
\begin{align*}
01011110 \\
10110011 \\
01111111 \\
11000001 \\
11100010 \\
10110101 \\
01111010 \\
11001101 \\
11111110 \\
10000011 \\
01000111
\end{align*}
\]

Figure 2.2: Sample cellular automata evolution
Some combinations may lead to constant configurations, some to simple oscillating patterns, and some (assuming an infinite length array) may go on forever. These three groupings correspond to the scheme now generally used to classify dynamical systems in terms of the appearance of their attractors in phase space as either a "fixed point", a "limit cycle", or "chaotic", respectively. It is possible to pose the halting problem to a given cellular automaton and one finds that for some it is decidable while it is undecidable for others.

The main points are that (1) cellular automata provide a prime example that local interactions can produce global complexity in a given system, that complexity can be built from the bottom up out of simple discrete elements and the collective behavior produces the phenomena we are interested in; and (2) because of this they provide a means to model complex physical systems which might be used in animation.

How can these ideas be put to use in animation? Consider a 2-dimensional automaton, with a neighborhood for each cell of corresponding dimension. Such automata have been developed for application to problems in image processing where each cell receives the input from the corresponding position of the image, and the cells cooperatively solve a feature extraction or pattern recognition problem on the given image.

In computer animation, we wish to reverse this process and have the cellular automata generate the image, animating it as it evolves over time. The game
of life first introduced by Gardner [Gardner 1970] served to popularize the idea of creating images in a “game-like” manner. [Toffoli 1987] is full of fascinating examples of images generated in this manner. These images display a wide variety of dynamic phenomena.

Cellular automata are being used to model physical systems. One of the more important uses is in the study of fluid dynamics [Lim 1988]. Lattice gas automata are models of cellular arrays in which each element represents a discrete volume in space (although symmetry is assumed along the third axis so that the simulation can performed more efficiently in 2D.) The state of each element represents the presence or absence of gas particles with velocities in one of several discrete directions. For example, an orthogonal lattice would represent particles traveling in only one of four directions (call them north, south, east, and west). An element can represent at most four particles, one traveling in each of the four directions. At each time step, particles are moved from one element to another according to their outbound velocity. When several particles meet in one cell, collision rules determining the configuration of the outbound particles when the time step is completed.

Clearly, the model is “stylized” in that all particles travel at the same speed, occupy only discrete positions in space, and only travel in one of a few discrete directions. However, with the proper choice of the transition function, the conservation
laws (of mass, momentum, and energy) which are captured by the Navier-Stokes equations used to mathematically model real fluids can be preserved in the lattice gas model. Although at the microscopic level the simplistic reductions are quite severe, at the *macroscopic* level many of the properties exhibited by the lattice gas model are identical to those of physical gasses. These models are capable of capturing such phenomena as waves, vortices and turbulence.

2.3.3 Actor-Based Systems

When the term *actor* is used in the context of a theatrical production, an image is brought to mind of an autonomous entity acting under the supervision of a stage director. Since animation as an art form descended directly from film and theater [Crafton 1982], the actor/director analogy applies here as well. In this case the animator acts as a director and the actors are the moving objects or systems being animated. This is the essence of actor-based animation.

Despite the obvious applicability of the term "actor" to computer animation as theatrical entertainment, the term also has origins in the field of computer science and related areas, such as simulation and artificial intelligence. The remainder of this section will trace the historical development of actor-based systems in computer science and animation. The intent of this review is to illuminate the employment of actor systems as a natural mechanism for abstraction to combat complexity in computer science, simulation and thus animation.
SIMULA: Classes, Inheritance, and Quasi-Parallelism

The history of actor-based systems can be traced back through the origins of object-oriented programming to the SIMULA language. During the development of programming languages in the mid 1960's there was a thrust to provide users with general capabilities to define their own application specific data and control abstractions. SIMULA, an ALGOL based language initially designed for discrete event simulations [Dahl 1966], introduced the notion of classes, wherein both the data and control abstractions could be defined in one syntactic unit. A class definition consists of a description of: (1) the data structure needed to represent an instance of that class, as well as (2) the sequential procedures needed to manipulate that structure. Here we use the term "instance" to mean an actual incarnation of a member of that class. Instances of a class are created dynamically by a running program as needed by allocating memory for the data structures and tagging them with the related class procedures. For example, a class representing complex numbers would have data structures to hold the real and imaginary coefficients as well as the procedures to add and multiply two complex numbers together. A simple program to calculate the sum of two complex numbers might create three instances of a complex number, two instances to hold the inputs and one to hold the result.

It is interesting to note that instances of a given class were originally referred to as "objects" in the programming sense of the word [Nygaard 1978], and that
this later led to the term "object-oriented". We reserve the use of the singular term "object" to refer to the imaginary body that an animator is attempting to set in motion.

The main advantage of the class model introduced by SIMULA is in the localization of the description of a user defined programming abstraction. Unlike prior programming abstraction mechanisms, classes allow the programmer to conveniently encapsulate both the structural and operational implementation of the abstraction in a single definition. Once defined, the programmer may then utilize instances of that class in terms of the abstract notions of the class properties and behaviors, regardless of the particular form of the class implementation. The programmer can concentrate on "what" an instance does, not on "how" it does it.

The localized nature of the class definition enhances the modularity and hence modifiability of SIMULA programs. A change in the implementation of a class need only affect the class definition itself. Furthermore, uses of instances of the class, if accessed solely through the procedures provided by the class mechanism, should be unaffected by such changes. Unfortunately, SIMULA did not force the programmer to use the class mechanism in this manner. SIMULA did not enforce the concept of data hiding, wherein the implementation of a class was inaccessible to external program units. Thus, an undisciplined programmer could inadvertently corrupt the data structures of a class instance, nullifying the advantages provided by using a class. Later we shall describe how SMALLTALK solves this problem.
SIMULA also introduced another programming feature in relation to classes, namely the ability to define one class (a subclass) in terms of previously defined class. This notion evolved into the modern object-oriented concept of “inheritance” where a subclass “inherits” the properties and behaviors (structures and operations) of its parent superclass. This is an easy way to make a variety of classes share some common properties. Suppose, for example, a program is to be written which manipulates linked lists of polar \((r,\theta)\) and Cartesian \((x,y)\) coordinates. A generic Linked-List-Member class could be defined consisting of a structure with forward and back pointers. The classes Polar-Linked-List-Member and Cartesian-Linked-List-Member could then be defined as subclasses of Linked-List-Member. Although each of these new subclasses is specialized for particular (polar or Cartesian) vector operations, they share in common the properties of members of a linked list. Generic operations for manipulating linked lists need be written only once and can be applied to both subclasses independent of the particular specialization.

Inheritance allows abstract programming constructs shared by several program modules to be factored out and encapsulated within the class definition. Using inheritance, a hierarchy of classes may be constructed. Each level in the hierarchy corresponds to a particular level of abstraction used during the program design. This aspect helps to keep programs simpler, shorter, and more tightly organized.
Another concept introduced by SIMULA and related to its origin as a discrete event simulation language is the notion of "quasi-parallelism". The designers of SIMULA took the view of a discrete event system as a collection of processes (class instances) whose interactions comprise the behavior of the system. This view implicitly holds that the processes are acting conceptually in parallel. The implementation of SIMULA allowed processes (class instances) to explicitly create and activate one another in an event driven fashion. This programming feature allowed one to increase the dynamic complexity of a program while still maintaining local control over that complexity within the class definitions themselves.

The three notions of classes, inheritance, and parallelism, introduced by SIMULA, provide a powerful framework for constructing simulations. They enable one to design the structure of the program to very closely match the structure of the system being modeled. Classes can be used to represent generic properties and behaviors of objects which inhabit the system to be modeled. Inheritance can be used to build classes which represent specialized or particular instances of these objects at several levels of abstraction. Finally, the simulation itself can be conducted by activating the objects so constructed in parallel. Hopefully, the interactions of these objects will reliably reproduce the behavior of the modeled system.

1The term "quasi-parallel" was used since the hardware would still be executing the compiled code in serial.
Object-Oriented SMALLTALK: Introducing Message Passing

SMALLTALK, the first true general purpose object-oriented language [Goldberg 1983], built upon SIMULA’s ideas of classes and inheritance and introduced the concept of message passing between class instances. As stated earlier, SIMULA did not hide the implementation of a class from the program units that used them, allowing the use of the instance to possibly depend upon its implementation. Message passing effectively solved this problem by forcing all references to a class instance through procedures which are defined by the class itself. These procedures (called methods) are invoked by sending a class instance the appropriate message. Since these methods define the only means in which an instance may be referenced, the instance’s internal integrity will remain secure. Furthermore, since the internal implementation of the class is hidden from the message sender (caller), program modularity is enforced; changes in the class implementation will not affect code which employs the class. Thus, the SMALLTALK class mechanism can be considered an abstract data type in the full sense of the term ([Ghezzi 1982] pp. 19) since it encapsulates and hides both the data structure and related operations.

SMALLTALK was originally designed not only as a programming language to be used on a powerful personal computer, but also as a complete software development environment: it contained features useful during both the design and the implementation phases of large programmed systems. The development of SMALLTALK was instrumental in illustrating the power of the object-oriented
paradigm to fully support the concept of data abstraction, and hence to provide a viable means to attack complexity in software systems development. In Section 2.3.3 we will introduce another paradigm useful in controlling complexity, namely, the actor-based systems.

Object-Oriented Simulation Systems

The advantages of the object-oriented approach are reflected in the number of such simulation systems developed and the variety of problems for which they were designed. A brief survey of several rule based object-oriented simulation systems is presented in [Adelsberger 1986]. His review supports the claims that the approach is intuitively more natural and convenient for representing simulated objects at multiple levels of abstraction. The paper ends with suggestions for improving these systems by providing graphical interfaces and improved run-time interactivity.

SIMULA and SMALLTALK, which were discussed at length above, were both general purpose languages well suited to performing discrete event simulations. In fact, in [Goldberg 1983], two classes, “Simulation” and “SimulationObject” are described for building such models. The “Simulation” class manages a collection of “SimulationObjects” to model such situations as the flow of customers through a bank, or cars through a car wash. In [Bézivin 1987] these classes are extended to perform experiments with simulations of this type in a distributed system. Bézivin
argues that his results support the use of SMALLTALK as a powerful environment in which to develop and test new simulation techniques.

Thinglab [Borning 1981] is a constraint-oriented simulation system written as an extension to SMALLTALK. Its primary goal is to "design a computer-based environment for interactive graphics simulations of experiments in physics and geometry" ([Borning 1981], pp. 354). Borning proposes using classes of objects to define libraries of building blocks for each simulation domain. For example, the domain of electrical circuits would consist of resistors, wires, batteries, and meters.

In a similar vein, Hollan et al. [Hollan 1984] have constructed an interactive, inspectable simulation of a steam driven propulsion system appropriately called STEAMER, to be used for training Naval engineers. Concerned with enhancing the user's understanding of a complex system, a graphical interface was constructed which consisted of a variety of iconic primitives such as dials, graphs, pumps, valves, pipes and electrical components. When each of these icons is "tapped" into the underlying simulation at the appropriate places, they provide the user with easy access to inspecting the simulation at several levels of detail. The object-oriented nature of the graphical interface and the graphical editor used to construct it, has proved flexible enough to be adapted to several other simulations concerned with parallel distributed models of cognition ([Hollan 1984], pp. 24).

Another object-oriented language, ROSS, developed at the RAND Corporation, has also been used to construct knowledge-based simulations. In particular,
distributed problem solving by groups of cooperating (or competing) agents have been studied in the contexts of air traffic control, and fleets of remotely piloted vehicles in combat situations [Steeb 1984]. In their report [McArthur 1984] claim that “an object-oriented style of computation is especially suited to simulation in domains that may be thought of as consisting of autonomous intentionally interacting components” (italics theirs [McArthur 1984], pp. 10). However, they also report that non-intentional events caused as side effects of deliberate acts, (such as a radar blip appearing as the result of a craft’s penetration of an air space) are not so “naturally” modeled as messages between objects. The penetrator should not need to send a message to the radar telling it that the craft has penetrated. This author would argue that this problem can be surmounted somewhat if the radar in general were to be modeled as by a single “hyper-radar” object (hyper meaning omniscient), which knows the positions of all craft, and which tells each radar object when a craft has entered or left its radar range. Unfortunately, this is counter to the idea of local interaction and control.

The foregoing survey is by no means exhaustive. It is presented as evidence in support of the use of object-oriented simulation systems because of the natural mapping between real world and simulation objects, the ability of objects to represent various levels of abstraction, and as models of distributed problem solving.
Actor-Based Systems: Origins and Impact

At about the same time that the seed ideas for SMALLTALK were germinating at XEROX PARC, Carl Hewitt was introducing the notion of Actors at MIT [Hewitt 1973]. In an actor-based system, a class definition describes a set of behaviors and a local knowledge base. Like SMALLTALK class definitions, these embody both procedure and state. However, the term actor includes the computational process which executes the behaviors and manipulates the knowledge base of a particular class instance. Thus an actor represents not only the procedure and state of a instance, but the process that runs those procedures as well.

Each behavior (or method) is a procedural description of the actions that the actor should take upon the receipt of a message that invokes that behavior. When an actor receives a message, it first decides whether it possesses a behavior that corresponds to that message, and if so, executes that behavior. While executing this behavior, the actor may manipulate its local knowledge base, create other actors, communicate with other actors, and/or return a response to the message sender. An actor-based system is one in which all elements of the system are actors and the only activity possible is the transmission of messages between them.

It is hard to draw concrete lines between object-oriented systems and actor-based systems. The salient features of objects and actors are quite similar. Both support data abstraction by encapsulating hidden implementations, both use message passing to communicate information and both provide for some means of
inheritance (also known as specialization or knowledge sharing). However, differences which do appear are rooted in the historical origins of each methodology [Pugh 1984]. While the original notion of objects arose as a means to facilitate procedural and data abstraction in systems development [Stefik 1986], actors were born from the need to represent decentralized forms of knowledge, and the processes which controlled them.

The development of the actor concept arose from a growing awareness in the AI community that modeling intelligent behavior as a single monolithic problem solving agent did not capture important aspects apparent in existing natural systems: namely, that much processing appeared to be performed in parallel and that information must therefore be distributed and communicated within the agent. The alternative approach proposed was to use actors (or similar structures) to model intelligent behavior as a collection of communicating specialized experts, much in the way a society of scientists might solve problems in research. [Chandra 1983][Lenat 1975][Hewitt 1977][Minsky 1986].

The use of actor models is expanding into other areas of computer science. With the advent of commercially available parallel multi-processing machines [Seitz 1985][Hillis 1987], research in distributed systems is of critical importance if computer scientists are to adequately take advantage of this new computational power. One researcher [Agha 1986], is striving to formalize descriptions of actor systems
in order to study expressiveness and efficiency in the context of concurrent computations. The major properties of the actor model that Agha proposes are as follows:

1. Actors can create other actors.

2. Actors run asynchronously

3. Actors are referenced by a symbolic address.

Each of these properties represents important features that concurrent computations executing in a distributed environment will need to possess. Together, they maximize the ability of the software to fully utilize the available computational resources contained in the underlying network of parallel machines upon which an actor system might run.

First, by allowing actors to create other actors, an actor-based system has the ability to dynamically modify its own level of concurrency. For example, an actor, given a large problem to solve, can potentially apply a divide-and-conquer approach, by creating actors to solve each sub-problem. In this manner, the distribution of the computation can be increased as the solution progresses.

Second, the idea that actors are modeled to run asynchronously is based upon the concept that within a distributed system, it is impossible to define a unique (linear) global time. This is true because of the inherent limits upon the speed of communication (message delays) between any two processors. This restriction
limits the ability of two actors to simultaneously know each other’s state. Thus, it is impossible for them to directly synchronize their actions.

Nonetheless, it is possible to construct systems which abstractly appear to run synchronously. For example, one actor could be designated to act as a “global clock”. All other actors wait for a “tick” message from the clock. When it arrives, they perform a predetermined set of actions and then return a “done” message to the clock. The global clock actor waits for all actors to be done before sending out another “tick” (this was proposed by Kahn and Hewitt in [Kahn 1978]). Clearly such a scheme will waste computational resources if some actors run faster than others: the faster ones will be waiting for the slower ones to finish. This problem is reminiscent of numerical solutions to sets of differential equations. If the set contains both stiff and non-stiff equations, the integration time steps must still be chosen small enough to ensure the stability of the solution of the stiff equations, even though the solution of the remaining equations would be stable with larger steps. In a loose analogy, the more quickly solved equations must “wait” for the slower stiff equations.

The third feature of actor systems is that the actors can be referenced by a symbolic address. This has two important ramifications for distributed processing. The first is that an actor reference is independent of the processor (the particular piece of hardware) upon which a particular actor is running. Thus hardware configurations are transparent to the description of a distributed application, and
changes in the hardware configuration while the application is running can occur gracefully without interrupting the service provided by that application.

The second ramification of this symbolic referencing of actors is that it allows for the flexible dynamic reorganization of the network of relations between actors. A specific example of this type of reorganization is provided by AI models of collections of cooperating agents acting to solve a common problem, as discussed in section 2.3.3. For example, a collection of remotely piloted vehicles with a mission to defend a particular airspace, may need to rearrange their cooperative relationships to meet changes in the nature of perceived threats [Steeb 1984].

**Advantages of Actor-Based Simulation**

One of the main advantages of using an actor-based approach is the ability to construct meaningful and useful abstractions. These abstractions come in two forms: those the user builds up into a mental model of the system being simulated (the meaningful abstractions), and the corresponding programmed objects (actors) written into the simulation software (the useful abstractions). The latter of these two abstractions is useful while constructing the simulation software because it allows the programmed object to encapsulate both procedural and declarative information. This programming abstraction has already been discussed at length above. The former abstraction is concerned with how the user views a system to be simulated and abstracts those features from the system which must be modeled
if the simulation is to produce meaningful results. Conceptually this occurs whenever a simulation model is constructed. However, using an object-oriented actor-based approach, there is a very tight, modular correspondence between the mental objects that the user manipulates conceptually and the programmed objects controlled in the simulation software. This natural correspondence, or mapping between a programmed object's meaning and its use, enhances the users understanding and control over the operation of the simulation software.

Discretization of continuous systems is often performed to reduce complexity by allowing solutions to the global problem to be solved at a simpler local level. This technique is particularly evident in physical modeling such as the finite element method (section 2.1.3), where the physical domain is spatially broken down into discrete structures with simplified, more easily computed, physical properties. More abstract examples are the attempts by the AI community to model intelligence as collections of cooperating agents. Using actors to represent these discretized elements follows naturally because individual actors can be used in a one to one correspondence to represent the discrete elements. In addition, it is possible to create a hierarchy of actors to represent the different levels of abstraction contained within the simulated model. For example, one actor might represent the highest level of abstraction and control the total collection of actors representing the discretized elements.
Finally, the utilization of an actor-based approach allows the simulation to accurately mirror any concurrency which may be inherent in the natural system being modeled. An actor-based approach also leaves open the possibility of realizing the simulation on a distributed multi-processing architecture.

2.4 Effects of Reductionism on Animation

Much of what has been discussed in the preceding sections has either explicitly or implicitly affected the use of the computer for generating animation. This section will attempt to illuminate those effects.

2.4.1 Actors, AI, and Animation

Developments in the field of AI have greatly affected the computer animation field in two ways. First, AI has provided the powerful actor programming methodology which has influenced the development of several computer animation languages, such as ASAS [Reynolds 1982], Director [Kahn 1976], and CINEMIRA [Magnenat 1983b], to name a few. Secondly, research in AI has been laying the groundwork for more intelligent animation systems, such as those proposed by Kahn [Kahn 1979], and Murtagh [Murtagh 1985] as will be discussed in the sections that follow.

The evolution of object-oriented programming languages and actor-based systems has been very influential in the development of computer animation languages. One of the main reasons for this influence has been due the programming advantages offered by this approach. Computer animators are typically interested
in finding ways to increase the complexity of the dynamic systems which they can represent and control. Powerful abstraction mechanisms, which are available in actor systems can be readily and easily employed by programmers to extend the languages to suit such animator needs. In addition, these extensions follow naturally from the animator's mental model of the world that is being animated. This is important in a field which is attempting to bridge a gap between art and science, making high technology available to an often non-technical user.

Many researchers have realized that in order to increase the utility of animation systems, and the complexity of the animations they produce, the systems will need to take over more of the burden of generating the motions of the animated models. This can only be achieved if such systems contain more knowledge about the models that an animator is attempting to control. For example, Catmull [Catmull 1978] describes problems encountered when trying to automatically generate the frames between two key frames in a 2D line hand-drawn character animation. Normally, there is a one to one correspondence between the lines in the two key frames, and interpolation can be used to generate the corresponding lines in each of the inbetween frames. Unfortunately, if the character is rotating with respect to the viewer (turns from facing stage right to stage left, for example) then certain lines may disappear and other lines may appear between the two key frames, representing different portions of the character passing out of or into view. Catmull
suggests that AI techniques might be needed to handle the lack of line correspondence between the key frames. Of course, one solution to this problem, is to use full 3D instead of 2D animation, and let hidden line algorithms solve the problem. Of course, this introduces a host of other problems associated with 3D animation, among them, the control of the 3D motion which would make the character turn.

Other researchers have also indicated that more knowledge about the objects to be animated must be embedded within the animation system itself. Demos [Demos 1982] envisioned an animated world where balls automatically fell under the influence of gravity or rebounded from collisions, and entire motion sequences could be generated by such simple phrases as “the cat jumps off the table”. In work towards these ends, Zeltzer ([Zeltzer 1984] pp. 117) clarifies that in order for goal directed characters to satisfy their user supplied constraints, world knowledge about their surrounding environment must be readily available to the motion planning software. He proposes the construction of an “environmental working set” in which such information can be organized for efficient run time retrieval.

The remainder of this section discusses the work of four groups which have been affected by the field of AI through the actor paradigm: Reynolds, the Thalmanns, Kahn, and Murtagh.
2.4.2 Ani and the Director Language

Kenneth M. Kahn, a student of Carl Hewitt’s at MIT, was one of the earliest researchers to have applied the actor paradigm to problems in animation. He developed an animation language called Director and an animation generation system called Ani.

Kahn’s research was concerned with constructing a computer model of aesthetic creativity. In his thesis [Kahn 1979], he describes the Ani system as a program that creates computer animations from story descriptions. The input to ani was a high level explanation of the action contained in the story. This description included the characters, their personalities, the interactions between them, and the scenes and events important to the story. These facts were considered to be information that was to be conveyed to the viewer by the animation. The activity of the characters was produced by making knowledge-directed aesthetic choices based upon the information suggested by the story description, and built in knowledge derived from common sense, perceptual, and ethological sources. The resulting output of Ani was in the form of a Director language script, which could be used to produce the final animated graphics.

The Ani program was implemented as an extension to the Director language. The Director language was written in Lisp and based upon the actor paradigm. All entities within the Ani program and Director (each character, scene, subscene,
and choice) were actors. Within each actor a data base of knowledge could be represented as well as an arbitrarily complex procedure describing the actor's decision making activities. For example, decisions as to how the story should be animated were handled by "choice point" actors. The choice point actors made decisions based upon suggestions supplied by other actors. These "suggestion" actors acted as knowledge sources, for example, by comparing character personalities (such as "who is faster, stronger, etc"), or by culling information from the story description itself. The choice point actors embodied procedures for combining and resolving conflicts between suggestions as well as a data base of the suggestions received so far. The data base of suggestions represented the current design decisions about the animated action that a choice point actor had made up to that time. As new suggestions arrived, they were examined in light of the current design and, if appropriate, incorporated into the database after conflicts with existing suggestions (if any) were resolved. In this manner, an initial rough design of the animation incrementally evolved into its final form.

By organizing the choice point and suggestion actors in class-type hierarchical fashion, the knowledge of the story and animation design could be "chunked" (organized) at the right level of abstraction. Also, because the knowledge was shared in an organized way, redundancy of storage was minimized. Kahn claims that the modularity provided by actors was essential in easing the costs of design, implementation, and debugging of Ani and Director.
Extensions to the Director language necessitated by Ani were facilitated by the
actor paradigm of the Director Language in two ways. First, Director had to be
extended to accommodate some of the features of Ani. Here the modularity and
locality of information simplified this task for Kahn. Secondly, one of Director's
tasks is to coordinate the concurrent activities of characters in a given scene. By
having all actors respond to explicit global "ticks" marking off time, all actors can
appear to march forward in synchrony. He refers to the achievement of apparent
parallel activity as quasi parallelism [Kahn 1978].

In comparison, the work presented here is not concerned with such an ambitious
goal as modeling creativity. We are interested, however, in what kinds of knowledge
might be needed to allow actors to cooperatively solve a constraint problem in
animation. We must agree with Kahn that the more sophisticated animation
systems of the future must contain more knowledge about the objects that are
being animated and the visual effects the animator intends to convey by displaying
those objects. In Kahn's domain of animation as story telling, such systems can
help plan animations if they understand the intended outcomes of such creations.

The Ani program uses the story description as information about how to plan
the animation based upon knowledge of characters, their interactions and the point
of the story. In effect, Ani tries to set up the initial conditions of the characters so as
to achieve the desired outcome of the story. Once the characters have been defined,
Director simulates their actions to produce the animated graphics representing
the story. His main claim ([Kahn 1979], pp. 50) is that designing animation is knowledge based problem solving, whereas the generation of the actual animation is simulation.

Our interest lies in exploring ways in which simulation can be better used by animators. Our intent is not to mimic the creativity of the animator, rather to supplement that creativity with new choices for producing interesting visual effects. In our approach, the user sets up the initial conditions between the animated objects, and the system simulates the interactions between them. The generation of the animation by simulation uses a technique of "ticks" similar to that described in [Kahn 1978].

2.4.3 Murtagh

Another researcher who can be credited with taking advantage of an actor-based approach towards computer animation is Murtagh. In his thesis [Murtagh 1985], Murtagh describes a knowledge-based, object-oriented language geared towards providing users with a level of intelligent control over animated objects. One of the primary design goals in Murtagh's system was that the animation system should understand the domain to be animated. His approach was to control complexity through abstraction and embedded world knowledge, and provide a degree of intelligence through the use of goals and multi-directional constraints. In the animated world, objects respond to their environments. Message passing is used to provide
each object with current information about its surroundings, in effect, simulating an object's "perception" of its environment. Complex inter-relationships can be created between objects through the use of multi-directional constraints, which allow the animated objects a degree of flexibility in choosing how to satisfy them.

The system developed by Murtagh is based upon PSN2, an object-oriented Lisp based language, originally developed to study knowledge representation in procedural semantic networks. PSN2 allowed for three abstraction hierarchies based upon the INSTANCE_OF, PART_OF, IS_A, relationships. These allowed for inheritance to be based upon classification, aggregation, or specialization abstractions, respectively. Although these language features provided a useful foundation upon which to build an animation language, Murtagh had to extend PSN2 in several ways in order to support animated object and motion representations. One of the first extensions was to add a message passing capability so that an actor paradigm could be supported. Two other extensions are the animated variables and the constraint maintenance capabilities.

Like the "newton" introduced by Reynolds in ASAS [Reynolds 1982] (and discussed in Section 2.4.4, page 64), or animated basic types of CINEMIRA [Magnenat 1983b] (discussed in Section 2.4.5, page 66), Murtagh has added animated variables which are symbols whose values change over time. The symbol has attached procedures which define how the variable is to change. For example, functions attached to an animated variable might linearly interpolate between two points,
generate a spline curve between a set of control points, and/or provide for "eases" (smooth acceleration/deceleration) between the points as the time parameter is linearly varied over the interval. The animated variable could then be used as a description of any property of an object over time; for example, say position, in which case the animated variable serves as a path description for a moving object. Thus, animated variables are used to specify the dynamic properties of objects.

Another extension that was added to the system was a means to handle constraint maintenance over time. Such constraints are called dynamic relations. The system itself is responsible for checking constraints at the appropriate moments, and raising exceptions when constraints are violated. The exceptions call user defined procedures to deal with the violation. It is the user's responsibility to supply the procedures which determine how a constraint violation is to be resolved. Some obvious constraints of importance to computer animators are interpenetration taboos, and such common sense notions as "unsupported objects must fall". Such constraints can become expensive to implement since each object may need to be checked each frame for violations. Murtagh proposes the use of a constraint dependency network to reduce inefficiencies of such implementations. When changes in relationship values occur they are immediately propagated through the network.

Inheritance properties available through the object-oriented approach, are used when constructing animated objects. All animated objects inherit attributes from
the class "WorldObjects". This class contains such common properties as location, physical mass, elasticity, contact/support list, as well as information regarding global constraints which are applicable to that particular object. One useful distinction Murtagh makes is between static objects which do not move (the theatrical analogy to "props" and "set"), and active objects ("actors") which move. Actors are further divided into subclasses such as lights, observer (camera), simple, or composite objects. The simple and composite subclasses capture the distinction between monolithic rigid objects vs. an articulated hierarchy of rigid objects such as a skeleton.

Murtagh's treatment of motion is interesting in that he attempts to represent motion by an explicit class instance, not as a byproduct of the system activity. The natural language processing flavor of the procedural semantic networking language (PSN2) upon which Murtagh's system is based meshes well with his definition of motion as an action verb, with the associated grammatical noun phrases. Murtagh represents motion by a class that contains subject, agent, and instrument slots. The subject is the object exhibiting the motion, whereas the agent is the object causing the motion. The instrument represents the object which is transmitting the force between the agent and the subject. During the simulation, actors which are assigned to the subject slots of a motion instance respond to the motion instance's message commands.
Murtagh differentiates between rotational and translational descriptions of motion as well as words which classify motion as either reflexive (relative to the object itself), or objective (relative to other objects). Consequently, the motion classes which Murtagh has defined in his system are: objective-directional, objective-rotational, reflexive-directional, and reflexive-rotational. These distinctions make it possible for a user to define a motion object such as “move left with respect to the camera” (objective-directional). This flexibility in motion specification allows for the camera position to be changed and still obtain the desired leftward motion of the object.

The simulation interpreter in Murtagh’s system is responsible for simulating the activity of the actors in parallel in order to produce the resultant graphical animation. Time is marked off in “clicks” which are equivalent to global “ticks” discussed earlier. Each click causes a standard set of procedures to be run to update all the actors to the next “current” time. Throughout the system, time can be represented symbolically so that events can be scheduled relative to other events in the system. This makes it a simple matter to edit the timing of one event and still maintain proper temporal relationships between events.

At each click, pending messages are forwarded to actors, and animated variables are updated to the current time. A global task manager is run which plans the satisfaction of all goal oriented objects. Motion objects are created, if necessary, to move the goal oriented objects toward their respective goals.
The next step is to non-deterministically serialize all the motion objects for the current click. Murtagh claims that this is important because if the motions are considered to be occurring concurrently, there should be no pre-determined order of simulation. However, there are certain situations where this non-deterministic ordering of motion execution can lead to scheduling problems: for example, where a moving body is moving towards a moving goal. If the body is updated before the goal moves, there will be a slight lag in its response to changes in the location of the goal. This may or may not be visually important in the final animation depending upon the grain size of the time steps. However, the operational differences are significant. Murtagh solves this problem in two ways. One way is to attach preconditions to the motion so that dependent motions are executed last. Another way is to simply allow the user to explicitly specify the execution ordering of the motions.

Once serialized, the subject of each motion is animated, thereby computing its state for the next click time. Next, constraints are checked and corrected if violated. Finally, relationships are made or broken, or new objects created as necessary.

The non-deterministic serialization of the motions in the system during the update cycle is pedagogically useful for demonstrating the theoretical point that all motions occur in parallel. However, in a practical system, when the motions are independent of order, it is an unnecessary step. If the motion executions are
independent of order, then no harm will be done if they are performed in the same order each cycle, and from a pragmatic point of view, it is more efficient to do so.

In contrast to Murtagh, the intent of the work presented here concentrates on exploring the use of simulation to visually enhance the motion complexity of animated scenes. One of the forces which drove the development of this work was to provide the capability to produce high quality, interesting, simulated motion for graphics. For example, if the visual essence of simple physical motions can be captured, they can form the basis for a library of useful motions. This motion library can then be used as a foundational layer upon which more sophisticated goal directed actor-based models can be built.

The motion classes proposed by Murtagh, incorporating translational and rotational motion, reflect the state of the art in current animation systems which tend to treat all objects as rigid monolithic bodies. They are insufficient for describing complex motion patterns of a dense population of moving elements, be they independent objects, or the discretized elements of a single object. These classes would be quite cumbersome if used to describe the motions of cloth fluttering in the wind, or swirling clouds of gaseous matter that might be generated by a using a particle system. Nonetheless, the explicit naming of motion types, if developed further, provides the user an important symbolic handle to explicitly manipulate motion. Such handles will be necessary for systems with natural language interfaces.
It is interesting that both approaches must resolve scheduling problems which can occur during the update cycle. This problem arises from simulating a continuous process by a discrete series of time steps. The scheduling problem disappears in the limit of infinitely small steps, since the size of the introduced error is directly related to the size of the step. Agha [Agha 1986] says this problem is also caused by von Neumann architecture of a common store, where state information is kept accessible to all processes, requiring synchronization to ensure the integrity of each data access operation. Agha’s solution is to bundle state information into messages passed onto an actor's successor (offspring) so that it is not tied to particular memory locations. The test-bed described herein does not address this problem and hence requires the user to explicitly specify the ordering of scheduled operations.

2.4.4 Reynolds

Reynolds developed the Actor/Scriptor Animation System (ASAS) as part of his thesis project while studying at MIT [Reynolds 1978]. Later it was integrated with scene simulation software already present at Information International, Inc. and was used in a commercial animation production environment to generate scenes for the Disney movie TRON [Reynolds 1982], [Disney 1982].

ASAS was heavily influenced by developments in AI research, particularly Hewitt’s ideas about actors and Kahn’s use of actors for animating stories. ASAS is written as an extension of Lisp, and uses the actor paradigm to help modularize
and localize the software which controls particular visual aspects of an animated scene. Reynolds likens his actors to "chunks" of code which are executed once for each frame rendered.

Since ASAS is an extension of Lisp, it is therefore a fully general programming environment; all features of both Lisp and ASAS are simultaneously accessible to the programmer. In addition, ASAS provides a set of standard geometric objects and their related operators commonly used in an animation environment; for example, lights, cameras, polygonal objects, and the usual methods for moving these objects about such as translations, rotations, and scaling. Reynolds introduced the concept of an animated number, called a "newton", which symbolically represents a value as a function of time. Typically, newtons are used to represent curves with certain properties, but they can be used anywhere a number would be used. Most often, they are used to specify the path an animated object is to follow. Between frames of an animation, the system automatically updates the value of these dynamic variables.

Scripts form the backbone of an animation specification in ASAS. A script is a named procedure responsible for producing an animated sequence. Scripts are used to initialize the animated sequence, to execute a series of subexpressions called animate blocks, and finally close the animated sequence when the subexpressions have finished execution.
The animate blocks enclosed within the script are analogous to the scenes of a movie. Each block is responsible for generating an entire scene. Animate blocks control the activity of a group of actors throughout a prespecified frame interval of a particular scene. An animate block is a loop which generates one frame at each iteration. Within the loop formed by the animate block different actors may be selectively activated (run) depending upon the particular frame currently being computed.

Actors are the procedural abstractions responsible for the visual appearance of each frame. When an actor is run, its old state from the previous frame is restored and the new state for the current frame is computed. This new state is stored (for future reference during the next frame), and relevant state variables that visually effect the scene are forwarded to the rendering software.

To this author, the most striking concept that Reynolds introduced was that of "Behavior Simulation" [Reynolds 1982]. This is a form of animation wherein the behaviors of a set of actors are procedurally defined, and a "micro-world" is populated with these actors. By running a simulation of this micro-world one obtains a glimpse of what would take place such an environment. This is a form of experimental animation in which the simulation produces an answer to the question "what would happen if" such a world existed. Reynolds further suggested the use of message passing to allow the actors to communicate with one another. Because message passing can model the interactions within a large group of actors, complex
scenes may be generated. It was Reynolds who suggested that message passing enabled the modeling of flocks of birds and schools of fish [Reynolds 1982]. In a subsequent paper [Reynolds 1987] he successfully demonstrated the application of an actor approach in just such a situation with the production of the film *Breaking the Ice*. His work has heavily influenced the concepts introduced in this research.

2.4.5 The Thalmanns

Another group that took advantage of the abstraction capabilities provided by an actor-based approach was the team of Nadia Magnenat-Thalmann and Daniel Thalmann. Their work was influenced by the success of Reynolds [Reynolds 1982] and appeared immediately thereafter [Magnenat 1983a]. In [Magnenat 1983b] they describe how their Mira-3D preprocessor implements extensions of Pascal to create abstract graphical data types, (which emulate certain aspects of the Simula class mechanism), and how these types are used in their Cinemira-2 animation sub-language.

Cinemira-2 is a procedural language for creating animation scripts. The main types provided are the animated basic types, the camera types, and the actor type. The animated basic types are extensions to the idea of the “newton” (see Section 2.4.4, page 64) introduced by Reynolds [Reynolds 1982]. These are standard types (such as integer, real, or vector) whose value is automatically updated over time.
The user must specify the starting and ending times and values, and the law (procedure) that controls their evolution over time.

The camera types control all the visual parameters associated with the viewpoint (GSPC Core standard is followed). Special procedures to allow clipping and coloring of windows, exist as well. These types make possible special effects similar to those created by an optical printer [Magnenat 1985b].

The concept behind an actor type is an abstract graphical data type that contains a procedural description of its own motion. The actor types are similar to the actors described by Reynolds [Reynolds 1982], in that they localize and modularize code which controls the motion of a graphical entity. Each actor declared in a script includes the time interval during which the actor “lives”, and may internally define animated basic types which can be used in the procedural description of its activity. In addition, actors can communicate in a limited form of “message” passing by exchanging “Signals” which control the start or ending time of their activities.

A novel use of actors that Thalmann introduces is the use of subactors to form a hierarchy of actors [Magnenat 1985b]. Such a hierarchy is useful for coupling the action of one controlling actor with several subservient actors. For example, to construct a car with four wheels, a car actor is created with four wheel subactors. The velocity of the car is then used by the wheel subactors to determine their rate of spin as the car travels along a surface. More complicated examples could be
contrived where the controlling actor coordinates and synchronizes the activities of the subactors. This concept has been used help integrate keyframe based animation with procedurally generated animation [Forest 1986]. Keyframe information is used to control most aspects of the animation, unless procedural information is available, in which case it over-rides the interpolated keyframe information.

2.4.6 Physically-Based Modeling

One useful type of information about an object which can help describe how that object moves is the physical properties of an object. Physically-based modeling is a new area of animation research concerned with using such information to procedurally generate the motion of objects. In an attempt to bring more physical realism into play when creating the motion of computer animated objects, animators are increasingly turning to the engineering and physical sciences for help. Its main thrust has been to attempt to take advantage of existing scientific techniques for analyzing physical structures to understand why objects move as they do. The analysis itself is only of pedantic value to an animator. However, the byproducts of this analysis are often simulation techniques developed by scientists to aid in the creative design of new mechanical structures or processes. It is these simulation based design techniques which the computer animator can most readily put to use.

Early attempts to achieve more realistic motion in animation were based primarily upon keyframe techniques. Because of this basis, such attempts treated the
symptoms of the unrealistic motion; they helped hide deficiencies inherent in the rather crude means of generating motion available. An example of such attempts is the development of the many spline interpolation techniques which allowed the animator to describe smooth motion paths over which objects could travel. It is quite interesting to note that the development of splines arose from needs in the shipbuilding and aircraft industries ([Mortenson 1985] pp. 98). A tool (called a "spline") consisted of a flexible strip which could be used by a draftsman to trace out a smooth curve between a series of non-colinear key points on a drawing. Because the spline behaves structurally like a beam, it forms a minimum-energy curve when constrained to lie upon the control points. The smoothness of this curve has a certain natural aesthetic appeal which may be explained in part by its physical origins. It is precisely this smoothness which is desired in motion, making the spline a useful animation tool.

In addition to controlling the path of objects, the animators also needed to control the velocity of the object over the path. The techniques of easing in and out were used to smooth out the apparent "jerkiness" of motion that would be caused by instantaneous changes in velocity.

Here again, the basis for easing in and out is primarily physical. One need only look at how traditional animators were taught their trade [Thomas 1981] to see that this technique attempts to visually mimic the effects of natural objects accelerating
or decelerating due to the action of natural forces acting upon them. Physically-based modeling techniques provide the animator with a means to automatically generate realistic motion. With this technique, the burden of work is shifted from the specification of motion to the specification of the physical nature of the objects and their situation. The direct control of the object’s position at each frame is sacrificed by the animator, in addition to paying an extra computational cost to run the simulation. In return, the animator achieves realistic motion with a vastly reduced specification load.

The current research that needs to be accomplished before physically-based modeling techniques can be fully utilized is to discover which techniques produce qualities of motions most useful in animation, and then provide the animator with an adequate means of maintaining control over the progress of the such physical simulations.

The appendix provides the reader with a review of the relevant topics in classical mechanics which are needed to fully understand physically-based modeling techniques and how they relate to motion generation in animation, as discussed in the section that follows.

The next section presents a brief introduction to current work in an area of physically-based modeling which has proved to be of some interest to animators, namely deformable models.
2.4.7 Flexible Objects

In terms of Schewpe’s classification [Schweppe 1984] the definition for flexible objects is anything that is not a rigid monolithic object. To avoid confusion with the term “plastic” as Schweppe used it, I will substitute the term continuous. Flexible objects come in two forms which I shall designate as articulated and continuous. Both forms attempt to model singular objects whose shape or configuration changes over time. Articulated models of flexible objects model an object as a collection of rigid elements (called links) connected together by joints. The joints constrain the degrees of freedom that the links, if disconnected, would enjoy. Obvious examples of articulated flexible objects would be the human skeleton, and the associated mechanical facsimiles, embodied in the form of a variety of robots.

Articulated models have been explored by a number of researchers. Badler [Badler 1979] was concerned with describing human motion to computers for research in work space design and reach analysis. Zeltzer, Girard, and Wilhelms have all developed articulated models designed for the animation of humans and animals [Zeltzer 1984], [Wilhelms 1987], [Girard 1989]. More recently, such articulated descriptions have been combined with dynamics simulations to animate general linked structures such as figures or chains [Hahn 1988], [Barzel 1988], and more elastic type objects such as trees [Isaacs 1987].

The continuous model of flexible objects refers to modeling non-rigid objects where the lengths between (any) two adjacent points on an object may vary in time.
Whereas an articulated approach can be used to model an object such as a bicycle chain, continuous models are capable of capturing the motion of substances such as rubber, taffy and silly-putty. If one is only concerned with the visual effect, then in some sense the articulated continuous distinction is merely one of resolution. If enough links are added to a chain, it can be made to appear continuous.

A comparison of these two approaches in light of the basic nature of computer animation reveals why the articulated models were historically the first to appear. Because the articulated models can be based upon redundant information inherent in the object itself, in general they require far less memory storage for both the static model itself as well as the control information which describes the object's motion in time. If the articulated figure is composed of links which are similar (all links in a bike chain are the same, for example), the instancing technique introduced by [Sutherland 1963] requires that only one link primitive need be stored. The entire configuration for a particular frame may be specified by describing the location and orientation (one transformation matrix) of one of the links (usually the "root" link) and then the relative joint angles for the remaining links.

On the other hand a continuous model usually requires the information on the location of every point in the body in order to completely specify its shape. (Obviously this can be reduced somewhat by appropriate parameterizations of the objects surface). Multiply this by the number of frames and an animation script will be quite large. It is interesting to note that simulation is a means to reduce
the need to know this information a priori by procedurally encoding the motion of the points in the body. The tradeoff of course is that time must be spent to calculate the new positions at each frame.

Rigorous development of both elastic and inelastic models have been developed by Terzopoulos [Terzopoulos 1987], [Terzopoulos 1988a], [Terzopoulos 1988b]. Pure elastic models refer to substances which may be deformed by the application of external forces, yet which return to their original undeformed ("rest" or reference) shape when the forces are removed. Inelastic behavior is characterized by materials whose shape is permanently deformed by the action of forces. Thus, the past history is important in determining the present response of inelastic objects.

[Terzopoulos 1987] appears to be the first solid work in the area of elastically deformable models based upon a mathematically rigorous model. Therein, the theory of elasticity is used to construct differential equations which govern the shape and motion of such non-rigid objects over time. The deformations of these objects are caused by the influences of external forces; when these forces are removed, the object returns to its its original reference shape. The key questions which that research offers answers to are: how can we represent the original reference shape (and measure the deformations from this reference shape) of such objects and how can we represent the shape restorative nature of these objects when external forces are removed? By answering these questions the motion of such objects may be realistically simulated. A deformed object contains within it the "potential" to
return to its original reference shape. In this sense the deformation of any elastic object (take a spring for example) acts as a storage for potential energy, which is released as the object returns to its original shape. For elastic objects which mimic Hooke's law, one would intuitively expect that the greater the deformation of the body, the larger the stored potential energy would be. This potential energy is stored continuously throughout the body, with local areas of greater deformation containing more energy. Now, as in all the physically-based modeling techniques, Newton's laws which relate forces (not energy) to motion, are used to simulate the action of elastic objects. It comes as no surprise then that the nature of elastic objects to restore their reference shape is the result of internal forces which arise directly from the deformation. In fact, the internal force at a particular point in the body can be derived from the potential energy stored at that point by a simple rule: the force is equal to the negative of the derivative of the potential energy at that point. Thus, if we can represent the stored potential energy of a deformed object we can determine the internal forces and resulting motion of the object. In order to represent the position of each point in the body over time, and therefore its shape, Terzopoulos et. al. choose to parameterize the shape in body coordinates $a = (a_1, a_2, a_3)$. The problem that is to be solved is to find $r(a, t)$ where $r$ is a vector function of the world coordinates of the point $a$ at time $t$. Clearly, the potential energy stored in the deformed body will be a function of the objects current shape as described by $r(a, t)$. Terzopoulos et al. suggest the use of the
metric tensor or first fundamental form to represent the shape of a 3 dimensional body. The metric tensor captures the shape of a solid by implicitly representing the distances between nearby points within that body. Two such bodies will have the same shape if their metric tensors are the same. Furthermore, the metric tensor is immune to rigid body motions, which is what is needed since rigid body motions do not affect the internal potential energy stored within the body. This metric tensor \( G \) can be used to represent the reference shape of the object \( G_0 \) and the current shape \( G = G(r(a, t)) \). The integral over the entire body of a weighted norm of the difference between these two tensors can be used as a measure of the total stored potential energy of the object. In order to find the internal forces acting at a particular point the derivative of this integral must be evaluated at that point. Since this integral is a functional (a function of the function \( r \)) variational calculus must be used to find the variational derivative.

The implementation described in [Terzopoulos 1987] for surfaces goes roughly as follows. The potential energy of deformation is approximated using a weighted matrix norm applied to the difference between the current and reference tensors. A simplified approximation to the variational derivative of this equation is made to represent the internal forces acting to restore the object to its original shape. The next step is to discretize the surface into an MXN grid, and derive a stiffness matrix \( K \) based upon finite difference approximations to the partial differential equations derived from taking the variational derivative. By representing the mass
at each node and damping at each node as pure diagonal matrices \( M \) and \( C \), respectively), the equations of motion may be written in matrix form as:

\[
Ma + Cv + Kr = f
\]  

(2.1)

where \( v \) and \( a \) are the first and second derivatives of the position vector \( r \), and \( f \) is the external applied force. By approximating the acceleration and velocity with second order differences (involving \( r(t + dt) \)), the equation can be algebraically rearranged to isolate \( r(t + dt) \) so that the problem is reduced to solving a matrix equation of the following form:

\[
A(t) r(t + dt) = g(t)
\]  

(2.2)

This equation is solved at each time step for the new position of each grid point \( r(t + dt) \) using a matrix method such as Gauss Seidel.

The speed of solving the above equation depends heavily upon the fact that the matrices involved are both sparse and banded. By organizing the discretization grid properly, one can minimize the bandwidth of the matrix, allowing the matrix to be compressed. Nonetheless, the order of the method is generally \( O(n^3) \) and \( O(n^2) \) for banded matrices.
CHAPTER III

Behavioral Simulation in Animation

3.1 Motivation and Goals Behind Behavioral Simulation

In contrast to the process of generating still images, the specification and generation of motion is a comparatively difficult task. Still in a state of technological infancy, the process is either dependent upon simple interpolation techniques, or otherwise ill-defined. Motion specification is a difficult task to do by hand, frame by frame, or even with interpolation techniques. I believe that this is true, even for a classically trained artist, without the use of specially designed tools because the presentation of motion information to a computer can be an abstract process: translating an image in the mind's eye, into the parameters of a procedure that algorithmically controls an animated model. This fundamental problem of motion specification has consistently limited the motion complexity of computer animated scenes.

The primary goal of this research has been to enhance the motion complexity of animated scenes. The ideas presented here were driven by animators' frustration
over being unable to easily generate complex motion. This frustration appeared comical in light of the fact that they were often utilizing powerful computing engines which were devoting little energy to motion generation. Naturally, these workhorses, if properly programmed, could relieve much of that frustration. The crux of the problem that this research attacks is to find ways in which computational power may be brought to bear upon problems in generating animated motion. Our attempts have been to make greater computational resources readily available to the animator in the hopes of lightening the motion generation burden. The intended effect of this solution is to facilitate the production of animation that has greater motion complexity than ever before.

Naturally, if new computational resources are to be of use to artistic enterprises, they need to be cast into a form which can be easily understood and utilized by non-technical artists. We assumed that new approaches to animation might be more easily introduced if the level of abstractions presented to the artists were closely matched by or would be minor extensions to those already carried in the artists head. Additionally, we suspected that the presentation of the control structures would best be made in terms of the style in which computer animators were currently accustomed, i.e. scripted animation, (so called “guiding level”, as defined by [Zeltzer 1984]).

These reasons forced us to take into account the animators task and mental model of the worlds animators commonly wish to model. Unless the planned
animation is of a highly abstract (non-realistic) nature, it is natural to assume that animators think of objects and motion in terms of their real world representations and physical properties. Even if the planned animation is to be non-realistic, the presentation to the artist of objects in terms of realistic properties commonly handled in ordinary every-day situations can be a useful level of abstraction from which extended properties may be creatively added. Thus, initial cases with which to test this methodology would center upon presenting physical objects and their common physical properties to the animator for experimenting with this technique. The key to the success of this research lies in providing a capability for building low level abstractions up into higher ones for ease of control and construction of user designed models.

3.2 Behavioral Simulation Defined

The main ideas behind the concept of behavioral simulation were introduced to the animation community by Reynolds [Reynolds 1982]. In his paper he suggested that a flock of birds could be adequately simulated by modeling the local interactions between each bird. This idea was later expanded and formalized in a subsequent paper [Reynolds 1987].

The work presented here attempts to broaden these ideas into a form which is useful for producing animated motion in general. The concept that drives the use of behavioral simulation in animation is that simulation can play a major role in
generating complex motion. We define behavioral simulation as a "bottom up" discrete approach to simulation because the complexity of a system is represented by a large collection of discrete interacting agents. It is the effects of these interactions that helps to generate the complexity.

Behavioral simulation views the system to be modeled as a collection of discrete structures called "elements" and the informative "relationships" which interconnect them. The relationships represent pathways over which information exchanged between elements. For a given element, the total set of relationships with other elements is termed the "environment" of that element. Each element has a current state and a procedurally defined behavior which determines what its motive response will be to current conditions in its environment. The motive decision making is based upon information which is locally available, not globally defined. It is the interactions at the local level which produce the complex, emergent behavior of the collection as a whole.

The procedure for applying this technique to a given system consists of three steps: reducing the system to a set or sets of elements, defining the behaviors of the elements and the nature of their relations, and finally using these elements and relation primitives to reconstruct a model of the original system and running it as a simulation.

The first step begins by viewing the system at the appropriate level of abstraction, and reducing it to a set of interacting components. By the term "appropriate"
we mean the common sense level from which one symbolically thinks about such a
system. This requires the division of the system into a collection of agents which,
in its entirety, can represent the system as a whole. This collection can be hetero-
genous with several classes of elements being defined. We expect, however, that
the number of classes would be kept small in order to keep the economy of the
system description high.

The second step is to describe the behaviors of the individual elements. Since
this behavior depends upon the relationships with their environments (other ele-
ments), this step includes defining what those possible relationships are for each
element. Implicitly, the definition of each relationship contains the nature of the
information exchanged, and the form of the exchange (whether it is explicitly re-
quested or passively received).

Finally, the system model is completed by creating a specific instance of the
simulation to be run. This means creating the elements and specifying the topology
of their connections which mimics the original system being modeled. Included in
this step is the specification of any initial state information needed to run the
simulation.

Consider the case where an animator wishes to construct a complex scene such
as Times Square at rush hour. The main moving elements in such a scene would
be the cars, pedestrians, and possibly paper litter blowing about the streets. At
the highest level of abstraction, let us consider the global motion of these objects.
Such motion would be affected by the traffic signals and the collision avoiding nature of the cars and pedestrians as they interact with each other. In addition, the motion of paper litter would be affected by the weather, especially by the wind blowing it about. If it rains and the paper gets wet, the character of its motion would be changed, not being so easily moved by the wind. The weather would also change the nature of the motion of cars and particularly the pedestrians who might prefer to walk close to the buildings and stay under awnings. The interactions as described forms a network of relations which is diagramed in figure 3.1. These relationships help dictate the decision making rules and environmental information that each element will need in order to compute its behavior at each moment during a simulation. For example, a pedestrian would have rules which would include such concepts as having a destination ("walk to the north side of town"), and obeying common sense rules such as: "obey traffic signals", "avoid collisions with cars", "walk on the sidewalk except when crossing the street" and "try not to trip over the litter".

We have assumed for the moment that the articulated walking motions of the pedestrians or the physical motion of the elements in the scene is a "solved" problem, handled automatically at lower levels of abstraction within the model. One could imagine that robotics models can built into the articulated figure models as has been done by Girard [Girard 1989], and then a goal directed super class built on top of this module to direct the global motion of the actors. The models of
Figure 3.1: Network of relations in Times Square
the cars might be constructed in a similar manner wherein a dynamics/kinematic model would handle the physical/geometric nature of the car bouncing and turning as it accelerates/decelerates down the street controlled by a cognitive model of the driver's decision making processes as he/she attempts to satisfy the goal of the destination while navigating around potential collisions.

Once this basic model has been designed, a specific instance of a particular street scene can be created and simulated. This activity would require the instantiation of the classes which define the pedestrians, cars, litter, traffic signals, and weather, and then making the informative connections shown in figure 3.1.

Similarly, in the case of a flock of birds, or a flexible surface, the behavioral approach would first view the system from its natural level of abstraction: a flock as a collection of birds or a flexible surface as a collection of surfaces each with a slightly different orientation. From here the complex system is described in terms of such individual elements, a single bird, a single surface element. The behavior of individual elements is defined which includes how that behavior is affected by the environmental relationships (external stimuli). For example, birds tend to follow and fly close to the other birds in the flock, and a flexible surface element will bend and stretch according to the position and orientation of neighboring surfaces. The information needed to drive these responses is obtained through the
relationships. Finally, a complete system to be simulated is arrived at when an organized collection of these elements is created, arranged, and initialized so that it is ready for the simulation to begin.

In short, behavioral simulation is a simulation-based approach which uses reduction to define a system and the interactions between the reduced elements to recreate the complexity of the simulated system.

3.3 Comparison of Behavioral Simulation with Other Approaches

The evolution of the concepts underlying the behavioral simulation approach were heavily influenced by those areas discussed in chapter II. It is instructive at this point to clarify the similarities and differences between behavioral simulation and those related areas in the hope of better illuminating the nature and intended context of this approach.

3.3.1 Behavioral Simulation and Behavioral Animation of Reynold's

Behavioral simulation is an offspring of ideas presented by Craig Reynolds in [Reynolds 1982] and [Reynolds 1987]. In his approach, the system is discretized into animated actors that control their own behaviors. Behavioral simulation attempts
to generalize this approach by providing a framework in which different types of simulation may be constructed, for example, both physical as well as cognitive models. In addition, a general means is presented to allow these models to be built hierarchically in the class structures and the simulation model itself.

3.3.2 How Cellular Automata Compares with Behavioral Simulation

Behavioral simulation has much in common with the body of work that is currently being done in the area of cellular automata. First of all, like behavioral simulation, cellular automata systems are concerned with the synthesis of abstract or concrete systems. Once synthesized, and implemented upon a cellular automata machine, experimentation and play can begin. The cellular automata universe is composed of a collection of discrete computing elements, whose behavior is completely specified in terms of a local relation. These laws are local in the sense that only nearby information effects the next state of the automaton. In addition, the laws are uniform; all automata follow the same laws. This uniformity naturally lends itself to parallel models of computation.

The similarity between cellular automata and behavioral simulation is uneven. First of all, behavioral simulation is concerned with the synthesis of environments and creating artistic expressions by experimenting within those environments. However, due to computational limitations, the need to streamline our abstract models and to represent different levels of detail in our models, causes us to allow
for different behavioral elements to be controlled by different rules. In other words, we allow the presence of heterogeneous environments populated by several different classes of elements, the behavior of each element being defined by the class to which it belongs.

Secondly, the simplicity of the cellular automata model, which allows for rigorous analysis of its computational properties, is a drawback when it comes to producing realistic images of physical phenomena. Nonetheless, the quality of the movement produced by cellular automaton simulations of physical phenomena in all its complexity is an effect that this research is interested in capturing. The way in which this quality is achieved is similar in that the rules of behavior governing the elements (the mass elements, for example) forces them to obey the conservation laws inherent in Newton’s equations, similar to the way the lattice gas automata models adhere to the conservation laws set up by the Navier-Stokes equations.

Thirdly, cellular automata usually represent a fixed grid in space. Behavioral simulation elements often represent concrete objects that move relative to one another, reflecting changes in either physical (connection or proximity) or quite possibly logical relationships. We prefer to see this characteristic explicitly represented by analogous changes in the data structures relating the elements. Similar arguments and concepts are captured by the “movable finite automata” presented in [Goel 1988].
Furthermore, when it comes time to actually implement apparently parallel operations on a serial machine, efficiency shortcuts may lead to lumped rather than distributed storage of data. For example, a collision detection model may require object location information to be reorganized into one data structure.

Both approaches suffer from the large computational resources required by the very nature of representing a model using a large number of components.

3.3.3 How Particle Systems Compare with Behavioral Simulation

Behavioral simulation models have taken advantage of particle techniques for modeling natural phenomena. For example, matter (mass) is represented as a particle since it is a convenient abstraction in which rotational properties can be ignored. However, the "primitive" elements used in behavioral models can be somewhat more complex than those used by Reeves [Reeves 1983] in terms of the procedures that govern the behaviors of those elements. As a result, the population size in behavioral simulations tend to be smaller with the complexity arising from the interactions between the elements. However, it should be noted that such interactions are not a priori excluded from Reeve's model.
3.4 Characteristics and Ramifications of Behavioral Simulation

Behavioral simulation differs from conventional approaches used in mathematical simulation in which an analytic equation is first derived to describe the system global behavior, and then a computer is used to solve the equation. The major difference is that behavioral simulation works from the "ground up", without the global description. The description of the behavior of the system is modeled at the microscopic rather than the macroscopic level. The global description is the simulated behavior which results. It is not a priori predicted behavior.

The need for constructing analytical equations globally describing complex systems arose in the past because such equations were the only convenient and tractable means to obtain a compact description of the behavior of the system. These equations allowed certain qualitative statements to be made about the system and even allowed the prediction of the system states over time in certain limited cases where linear assumptions could be made based upon "ideal" situations. The linearizing assumptions have been one of the reason for the success of technology over the past three hundred years. Unfortunately, most natural systems exhibit non-linear behaviors. Such behavior often leads to untractable equations for which no analytic solutions exist. This has limited our ability to study complex systems.
The use of the computer to perform simulations is changing the way we do science because of an expanded capability we now have to model and study complex systems. We are no longer bound to study only those systems for which we can write down equations. Now we can study systems for which we can write down algorithms which describe their behavior as well. Furthermore, these algorithms may pertain to the components (the elements) of the system and not the system as a whole. In this form, the description of a complex system in algorithmic form may in fact be simpler than the set of analytic equations required to capture that global complexity, because the complexity is expressed implicitly by the nature of the interacting components.

We claim that a discrete “bottom-up” approach has several advantages over attempting to model the global behavior using a top down analytical one. Let us define the analytic approach as having an explicit equation parameterized by time for the global behavior of an entity. Let us define the discrete structured approach as representing that same entity as a collection of discrete interacting elements. For example, to represent a flag waving in the wind, one might construct equations using trigonometric functions to represent the ripples, or one could use a finite element approach that modeled the flag as a series of adjacent “sub-surfaces”. This dissertation makes several arguments to support the discrete approach.

First, the discrete approach is a superset of all analytic approaches; numerical methods for solving analytic equations assume that in the limit of infinitely small
elements, the discrete equation becomes equivalent to the analytic equation.

Secondly, analytic solutions are able to describe only a very limited set of systems, usually the linear ones. A not too extreme statement might be that if these systems are describable, they are probably uninteresting visually. Naturally enough, animators wish to see more interesting, non-linear systems, which have no analytic solutions. For example, N-body problems where n is greater than 2 admit to no general analytic solutions. These systems are nonlinear and chaotic, subject to sensitivity of initial conditions.

Thirdly, the discrete structured approach is more robust. It is capable of modeling phenomena at several levels of abstraction, and gracefully allows drastic changes in system structure to automatically result in modified system behavior, i.e. such simulations are dynamically adaptive. This differs from some analytic approaches which may require a reformulation or even restatement of analytical equations if a new system configuration is to be handled by the model. Implicitly, this reformulation requires the system designer to be able to predict in advance all the types of situations which will arise. Even for a relatively simple system with a few interacting factors, this could mean a potentially exponential explosion of possible factor combinations which must be predicted and mathematically described. If we are to truly capture the nature of complex systems, we cannot for all such systems predict in advance all the effects of interactions between different system components. We argue that behavioral simulations which are built
from the bottom up can model behaviors and events which are unanticipated by the creators of the simulation. By modeling the local interactions between elements, global "emergent" phenomena may appear. Often, these visual effects are desirable, and help add to the visual appeal of the dynamic image.

It appeals to one's intuitive sense that a discrete structured approach would produce more complexity simply because the collection of elements represents more highly detailed state information about the system than can be captured in a single "global" equation. This does imply that more storage and processing is performed using this technique. However, the homogeneity of each class makes the technique easily adaptable to parallel programming approaches. Behavioral simulation more accurately mirrors the natural concurrency that occurs in the phenomena being modeled.
CHAPTER IV

Test-bed for Behavioral Simulations

In order to support experimentation using behavioral simulation as outlined above, a behavioral test-bed was constructed. The purpose of this test-bed was to facilitate the creation of behavioral models by the user and to provide for the generation of animation as output from the simulation. Since the possible types of simulations that a creative animator might wish to construct are unknown before hand, it was necessary to make the design as general as possible. This meant that the test-bed would need to serve as a rapid-prototyping facility for developing behavioral simulations. The remainder of this section presents an overview of the test-bed design and a more detailed view of the software constructs used to implement this design.

4.1 Test-bed Design and Implementation Overview

The development of the test-bed was driven by a number of factors which affected its design. Since the test-bed was to serve as a general framework for constructing and running behavioral simulations, the capability for dynamically creating
elements and relationships within the simulation framework was needed. Both of these types of structures must be easily accessed by the user so that their behavior can be observed during the simulations and modified between (and possibly during) simulation runs. Since behavioral simulation is couched in the idea that the elements respond to their environments, the simulation framework must also allow for the elements to obtain information about their environments. It also must provide a mechanism for evolving a system over time. In addition, elements may be historically sensitive, thus, some means must be provided for the elements to access their history of past states. Since our goal is to produce animation, a means must be provided allow the animator to exert arbitrary influences on the environment, and finally, so the user can view the results, a mechanism which serves to map the state of the elements onto a visual representation must exist.

Figure 4.1 shows an overview of the system components. The user interacts with the system through the top layer components which consist of the browser editor, the selection menu and the display window. The display window is used to display the current state of the animation as well as play back an animation at normal speed once a simulation run has finished. The animation is stored as a series of bit arrays in memory, which can be saved permanently on disk. Changes are made to the simulation database by "browsing" the elements and modifying their states. Finally, global operations are performed through the menu. One
such operation is to select a current environment from a set of demonstration environments. Other menu selections are used to start a simulation run.

Figure 4.2 shows the layout of the interaction window. The window consists of four panes: display, menu, message and scroller panes. The display shows the animated scene. The scroller allows the user to conveniently inspect a computed animation frame by frame, and to critically view the motion of any portion of the animation. The message window produces status reports during animation computations as well as during menu operations. The menu selection list shows the possible operations available. The SELECT option allows the selection of one of a number of demonstration environments to become the "current environment". These demonstration environments have been supplied with the system. Once selected, the EDIT option allows the user to browse all the elements in the current environment, possibly editing their initial states. The DISPLAY option tells the current environment to draw itself upon the screen. The INITIALIZE option sets the environment to its initial conditions and the GENERATE option begins to run a simulation. Once completed, the PLAYBACK option plays the stored frames back at normal speed. The LOAD and STORE options allow the animation to be permanently stored on disk.
Figure 4.1: System components
Figure 4.2: Interaction window
4.2 The Advantages of an Object-Oriented Approach

The complexity of these design factors was aided by modern software methodology. Actor-based object-oriented programming techniques were used to construct the test-bed, which consists of a set of programming constructs (classes) which supports the behavioral simulation methodology. The object-oriented approach greatly aided the software development task for a number of reasons. First, behavioral simulation is well suited to using an object-oriented approach. The services of message passing, class description, inheritance and data hiding readily supported by this approach form a natural means to represent elements which "own" their own behaviors. The behaviors of the modeled animated objects can be easily defined as the procedural properties of a class and its inheritance. In rough fashion, we construct a class for each behavior we wish to describe and add it to our library of behaviors. This modularity allows the user to treat behaviors as "black boxes" whose internal implementation remains hidden. The mechanism of class inheritance makes it easy to construct new behaviors by simply combining old classes or by building new classes on top of old. Each class serves as a template for creating an "instance" of an element (an actor). The class definition along with its inherited definitions describes those properties and behaviors that the instance will possess. Groups of elements are created by stamping out multiple instances of a class. Thus, using an object-oriented programming paradigm resulted in software that
is well structured and easily modified, with the various programming abstractions matching simulation model abstractions straightforwardly.

4.3 The Environmental Hierarchy

In order to facilitate the users access to the elements and relationships as well as the organization of environmental information, all elements within the test-bed are named and allocated in hierarchical fashion. The hierarchy is a tree structure with arbitrary branching at any node. The name serves as an address for the element, which can be specified either relative to the current element or absolute within the entire environment thus simulating the Unix \(^1\) file naming scheme. The basic commands for adding and retrieving elements from this hierarchy are the PUT and GET messages respectively. At a given node, pointers to all children are kept in an extendible array. The names for these children are stored as hash keys in a hash table that maps the name to the array index where the corresponding pointer to that child is stored (see figure 4.3). For nodes with a high branching factor, the hash table improves the retrieval speed.

The naming scheme uses special characters to signal the location of the element within the hierarchy. The “down-character” > and the “up-character” < signal corresponding movement within the hierarchy. Thus a request (GET 'cars>car2>driver') will cause the receiving element (say RCOT in figure

\(^1\) Unix is a trademark of AT&T Bell Laboratories
Figure 4.3: Environmental hierarchy node implementation
4.4) to look for an immediate child named cars. Once cars has been retrieved it in turn is sent a message to (get 'car2>driver'). In this manner, requests for information or actions can be propagated about the hierarchy until the proper recipients are found to handle them. Another example is that names beginning with the “down-character” are considered as absolute names and are thus propagated up to the root of the tree that then handles them as local requests (by deleting the leading “up-character”). For example, any node retrieving the request such as (get 'cars>car2>driver') will automatically send the message (get 'cars>car2>driver') to its parent node, if such a node exists. If a given node has no parent, then it must be the root node and it then handles the request as (get 'cars>car2>driver'). In similar fashion, requests beginning with the “up-character” are passed to the parent node. Thus the request (get 'cars>car2>driver') would cause the message (get 'cars>car2>driver') to be sent to the parent node, the leading “up-character” having been removed.

This hierarchical arrangement is designed to allow a simulated environment to be organized according to the levels of abstractions which can be mapped naturally from the real (or the users conception of the “real”) system. This facility is important because it allows the properties and behaviors of a system to be embedded within and executed from the level in the hierarchy which corresponds to the appropriate level of abstraction. For example, a useful property of physical
Figure 4.4: Example environmental hierarchy
systems is the center of mass. If a parent element is used to represent a collection of mass particles (represented as its children) then a center of mass calculation is appropriately handled by the parent. For example, the PHYS node in figure 4.4 might represent the physical model of CAR2 as a collection of mass points, each mass point being a child of PHYS. The center of mass calculations would occur within the PHYS node. In this manner, the hierarchy allows the simulation to execute at multiple levels of abstraction, which this author feels lends itself well to constructing robust simulations.

The hierarchy also allows us to partition the computations for efficiency. If two groups exist together in an environment but do not interact (there exists no relationships between them), then the behaviors of the groups may be computed separately. Although not implemented in the current system, the existence of the hierarchy paves the way for performing levels of detail selection during the simulation calculations. For example, if it can be determined that a collection of mass particles can be effectively represented by a rigid body, then a rigid body behavior may be executed at the parent level of detail, effectively bypassing the need for individual particle dynamics calculations to occur at the child level.

Finally, the hierarchy allows for the efficient propagation of messages. A message sent to the root of the hierarchy may be routed only to those children capable of responding to such messages, and the message can be intercepted and the response generated at the level of detail appropriate to the required resolution of the
scene.

It must be pointed out that the environmental hierarchy is independent of the network of relationships between elements which affects their simulated behavior. The hierarchy is provided for convenience of reference, organization and efficiency considerations. However, it is possible for the network of relations to take advantage of this hierarchy if the nature of the simulation lends itself to a hierarchical organization. In fact, it is usually the case that each environment is a hierarchy of elements, in which one parent element serves to represent a group of child elements. This allows the user to conveniently reference a collection of elements as a whole. For example, it is often convenient to group elements into the animated objects they represent and define relationships between objects at this “object” level, even though the relationships must really exist between the child “sub-elements” during the simulation run. If we make the global level “super-elements” intelligent, they can be used to simplify the management of the properties and relationships of the members of the group. For example, when a relationship is made between two animated objects at the super-element level, the relationships between the elements which compose each object can be automatically generated.

Finally, the hierarchy supports the “transformation tree” commonly used in graphics operations to locate the animated objects in the world. However, since we usually rely on the simulation to generate the motion, the transformation hierarchy is used only to specify the original locations of objects.
4.4 State History

Another property that animated elements will need is the capability of referencing their past histories. Since the test-bed was developed to experiment with different simulation models, we could not know a priori which information within the state description of an element would be dynamic and which would be static. Therefore, the entire state of the element is saved. The "historical" class which implemented this capability was constructed so that when instantiated, the element was duplicated and entered into a "historical" ring. In this manner, only a limited number of past states were saved. This ring structure also provided a place to store the results of calculations which were updates to the state at the next time step. In other words, the last item on the historical ring was destroyed and used for the next current state of the element.

The "historical" class in our class library is endowed with the unique property that when any instance that inherits from this class is created, it immediately makes two additional copies of itself. This feature is incorporated directly into the init message handler, the init message being automatically called whenever any class is instantiated. In order to prevent an infinite recursion of self creation upon receipt of this message, a flag is set in the argument list to turn this feature off when subsequent copies are created.
The "historical" ring structure is implemented as a doubly linked list, each element having a forward pointer to its "future" and a backward pointer to its "past". Because static variables which do not ordinarily change during a simulation are redundantly stored in each historical copy, complications will arise if the values of these variables become inconsistent. To ensure that each copy of these variables is always the same, the message handlers for setting these variables were modified as follows. Whenever any element of the ring structure receives a message to set the value of one of these static variables, the remaining elements in the ring are notified of the change. Although this is inefficient for setting the values of these variables, it leads to better efficiency at simulation run time because the values are stored locally. The alternative of storing these values in a common area would require indirect accesses during simulation run time, resulting in a slower simulation execution.

There are two advantages to having the past historical states of elements in actor form. First, a reference to previous state information can be obtained using the same message format as that used for current elements in the present. This simplifies the code. It also means conceptually that the object "still exists in the past" and thus when debugging a particular change of state, the past element may be directly examined. A second and related idea is that element states are retained in the same relation to their environments as when they determined their future behavior. Thus, to re-run and experiment from some previous state, one
merely resets the ring to that time period. (Of course, if previous historical data is required beyond the current depth of the ring, then the ring must also be restored to its previous historical state).

4.5 Scripting the Properties of Elements

In order to provide the animator with arbitrary control over a simulation, the facility for creating scripts was provided. A script is simply an element which produces a value (or values) as a function of time. By setting up relationships between scripts and arbitrary properties of elements, it is possible to directly control those properties during the course of a simulation. Since the behavior of the element is determined in part by its property values, its behavior will be affected by the script. As a consequence of the interactions between the scripted element and its environment, the global behavior of the environment may be affected as well.

Three basic script types are provided: scalar, vector, and transformation. Scripts are stored offline in ascii files. When a script element is instantiated, the ascii file is read and stored in an array. The script element is stored in the environmental hierarchy like any other element.

The creation of a script is independent of how it will be used. Therefore, a mechanism is needed to provide for transmitting a script value to a particular property of an animated object. Instances of the “data-inject” class are used to perform this task. A data-inject element is attached to a script and...
being scripted. A data-inject instance responds to the DATA-INJECT message by requesting the script value for the current time, possibly transforming this value, and then downloading this transformed value into the scripted instance. Figure 4.5 shows an example of the organization of a vector script being applied to some vector property of an actor.

The data-inject class is general purpose in that it can arbitrarily copy one environmental value into another at each step of the simulation. One of the advantages of having it in the environmental hierarchy is that it can be transformed like any other object in the environment. In this manner, a single script may be used to control the motion of several objects which must appear to move in parallel. Each object can be assigned a data-inject element to map the script values into the positions of the objects. If the script is centered about the origin then the script can be thought of as a script of offsets from some original location. Since the data-inject elements are transformed along with their associated scripted objects, they can apply this transform to the scripted values to determine the position of their scripted objects at each frame (see figure 4.6).

4.6 Stepping Along in Time

In order to move the simulation along in time, each element must determine its future state from some combination of its current state and information gathered from its environment. Since such calculations are often dependent upon the value
SCRIPT ACTOR
YIELDS INTERPOLATED VALUE
PARAMETERIZED BY TIME

F(TIME) = XYZ TIME

ASCII SCRIPT
XYZ
XYZ
XYZ
XYZ
XYZ
XYZ...

CURRENT TIME
XYZ

TRANSFORM

DATA INJECT ACTOR

X' Y' Z'

SOME PROPERTY: X' Y' Z'
"SCRIPTED" ACTOR

OCCURS DURING DATA INJECTION STEP

Figure 4.5: Vector script example
Figure 4.6: Transformable script
of the current or elapsed time, procedures for stepping along in time are built into
the environmental hierarchy in the form of simulation messages handlers.

The simulation message scheme keeps track of the time steps during the simula-
tion, sending the appropriate synchronization and display messages at each frame.
Simulation time is divided, naturally enough, into frames. Frames are further di-
vided into subframe time steps. These sub-frame time steps are used to achieve
apparent smooth motion even though the motion is calculated in discrete steps.
In this manner we can effectively integrate differential equations of motion, should
the motion of an element be described as such. During the simulation the root
of the environmental hierarchy executes the following pseudo code to move the
environment along in time:

INITIALIZATION

LOOP FOR ALL FRAMES

SUBFRAME TIME STEP LOOP

RENDER

END LOOP

The initialization step tells all elements to reset their states to the initial condi-
tions (initializes positions and velocities, rewinds scripts, etc.). The subframe time
step loop, executed in conjunction with error control schemes (adaptive step size),
moves the calculations along at a slow enough pace so that numerical instabilities
remain under control and the motion appears smooth. Finally the render step
computes and stores a single frame of the animation.

The subframe time step loop consists of the following pseudo code:

```plaintext
LOOP FOR ALL SUBFRAME TIME STEPS
    AFFECT
    RESPOND
    DATA INJECT
END LOOP
```

The "affect" and "respond" operations act like a heartbeat of transmitting and
receiving information. During the "affect" stage, all elements which affect their
neighbors must act. Consider a physical simulation for example, in which mass
particle elements respond to other force producing elements in the environment.
In such a simulation, all elements which produce forces would compute them based
upon the current state of the environment and transmit those forces to the mass
elements. During this phase, the mass elements would passively accumulate these
forces. The "respond" stage requires that all elements which were affected in the
previous step now collect information from their environments and respond to
the information received. During this stage, for example, the mass elements would
compute their new velocities and positions as a result of these environmental forces.

The "data inject" message is like the "hand of God", reaching in and magically
setting certain chosen state variables, or copying information from one location
to another. Its purpose is to enable the user to monitor or arbitrarily script any state variable. During this phase, any active script elements overwrite their current script values into the scripted element variables. This stage is also used to detect collisions and enforce interpenetration rules.

The data inject step was implemented as a separate step so that element computed state variables would be available for monitoring or transmission, and that user defined script values would override these element computed values. By separating this operation from the normal behavior of the elements, an element need not be aware of when it is being scripted; i.e. no special "scripted-actor" procedures need to be written, and the element computes its behavioral state as normal. This allows the user to arbitrarily invoke a script at any point during the simulation run.

4.7 Visual Representation

In order to maintain generality, the environmental description of an element is maintained independently of its visual representation. However, in order to graphically manipulate an element, it is required that an element or some hierarchical ancestor of the element be capable of representing an element's state in visual form. This flexibility allows the visual representation to take one of possibly several different forms. For example, if the sub-element of an element is to be explicitly manipulated, its graphical form may include additional graphical information than
when it is displayed as part of a hierarchical ancestor element. In effect, this allows for different "levels of detail" to be incorporated into the mapping of an element's state into the visual form. This concept is captured by the "visually represented" class definition. This class keeps a pointer to the visual representation of a particular element which may or may not be actively displayed, depending upon the context in which the element is currently being manipulated.

4.8 How the Test-bed was Implemented

The test-bed was implemented in Symbolics Common Lisp using the object-oriented Flavors dialect. The symbolics provided an excellent programming environment for rapid-prototyping the test-bed software.

4.9 How the Test-bed is Used

This section describes how the test-bed is used to construct experiments in obtaining global behaviors from groups of locally interacting actors. There are roughly four main steps: constructing the actors, constructing the environment, specifying any user required control mechanisms, and finally running the simulation experiments.

The first step is primarily a programming task of building an actor up from the base control structure flavors, and previously defined behaviors. This task includes specifying an actor's properties, state variables and kinds of relationships it can
have with other actors. The programmer must describe how the actor changes its state in response to other actors and how the actor will convert its current state into displayable form (usually this simply means rendering an icon at the actor's current world position).

The creation of an environment is both a programming task and a user task. The programmer defines what types of actors are to appear in the environment and the user defines the exact numbers of actors, their initial state values, and the topology of their interrelations. The user is responsible for defining any control mechanisms that are to be applied to an actor during the simulation. Currently the only direct control mechanism available is a "script", which allows the user to explicitly set an actor's state variable at each time step of the simulation. The script value loaded into the state variable during the data inject phase of the simulation loop, overrides any computed values that the actor itself may have placed there, effectively removing some of the actor's autonomy.

Developing an animation is the most enjoyable aspect of using the test-bed because one never knows exactly what the outcome will be. This process is usually an iterative one. The animator will generate a few seconds of animation, playback the results on the interaction window, edit the environment to change some property of an actor or script, and then rerun the simulation. Actually running a simulation experiment may take anywhere from under a second to a minute per frame depending upon the complexity of the environment (linear with the number
of actors). Obtaining quick feedback is important. For this reason, results are displayed as they are calculated, and the user can scroll through the computed frames while the simulation is running. In addition, the user may stop the simulation, edit the environment and then continue the simulation from where it left off, rather than start from scratch. In this manner, the environmental effects of sudden changes in an actor's state may be observed.

The remainder of this paper discusses how the behavioral test-bed played a role in development of a model of flexible objects.
CHAPTER V

Experimental Results

This section describes several behavioral simulation experiments which were designed and run in order to test the utility of the methodology in modeling complex systems and the production of interesting visual results. There were three tests performed. The first was the construction of a collection of simple goal directed actors: each actor moved from an initial starting position until a stationary goal position was achieved. The purpose of this test was to see if simple rules of motion could induce the effect in the viewer that these objects were “self-motivated”. The second test, which was suggested by the results of the first test, was to produce visually realistic motions of physical objects, in particular, flexible bodies. The third test served to unite the themes of the first two tests. A self propelled “moth-like” creature was constructed which possessed self motivated behaviors but was built upon a physically realistic model.
5.1 Goal Directed Motion

Goal directed motion has been studied by a number of researchers in robotics, artificial intelligence, and animation. Of particular relevance to this thesis is the work of Zeltzer [Zeltzer 1981] in which he describes complex motion systems which are goal driven from above and use environmental feedback to measure goal achievement.

The first experiment performed with the test-bed was to construct simple goal directed "arrows" that could, by controlling their own motion, move to a destination position from some random starting position. The aim of this experiment was to determine what rules and procedures were needed to make these "creatures" appear convincingly "alive".

A class was constructed within the test-bed which inherited the basic properties of position and its derivatives. In addition, this "goal-directed" class contained a state variable that represented the goal position for that instance as well as procedures for adjusting the current velocity (speed and direction) so as to move towards and achieve that goal position.

Initial tests revealed several important aspects to producing "life-like" motion. First, physical laws must be used to make the motion appear realistic. (This realization influenced the choice of the second set of experiments in physically based modeling.) Physical laws allowed the goal directed creatures to smoothly
accelerate towards their goal. Of course limitations were placed upon the maximum acceleration and the maximum velocity achievable by a particular instance. These limitations were necessary so that these creatures did not instantaneously (and unrealistically) achieve their goals. Finally, procedures were added to make the creatures always face in the direction of their motion and to smoothly decelerate as they closed in on their destinations.

Although this first test involved an extremely simple model, it served to illuminate those aspects of procedurally generated motion that are important to the principles of traditional animation [Lasseter 1987]. For example, the orientation of the “arrow” creature towards the goal anticipates the achievement of that goal, as does the deceleration as the goal is neared. Also, smooth acceleration, or easing in and out, needs to be simulated using physically based models.

5.2 Physically Based Model for Flexible Objects

The development of the model for flexible objects grew from a desire to provide animators with a means to easily generate realistic motion based upon natural physical laws. Since physical laws provide simple rules for behavior, they might provide a useful first test of the system. A collection of physically behaving actors might produce interesting large scale phenomena.

Newton’s first two laws indicate quite clearly that an important property of an object is its mass and that its motion is affected by the forces acting upon it. Thus,
if we wish to use mechanical simulation to generate realistic motion then we must use primitive actors which capture these notions of inertial masses responding to environmental forces. Our approach, then, is to populate the system with mass particle actors and the force actors relating them.

5.2.1 The Mass Actor

The mass actor represents matter at a point. It has the properties of mass, position, and velocity. Newton's first and second laws determine the behavior of a mass actor in the presence or absence of forces. This representation simplifies the dynamics calculations since rotational dynamics of a point mass can be ignored.

At each time step the vector sum of all forces acting on a mass is determined. The acceleration is determined using Newton's second law and the velocity and position of the mass in the next time step is calculated using the difference formulae shown in equations 5.1 and 5.2. Note that the first law holds in the absence of forces.

\[ v(t + Dt) = v(t) + a(t) \cdot Dt \quad (5.1) \]

\[ x(t + Dt) = x(t) + v(t) \cdot Dt \quad (5.2) \]
Where:

\( Dt \) - “delta t”, time step duration.
\( t \) - time of current sample
\( t + Dt \) - time of next sample
\( v(t) \) - velocity at time \( t \)
\( x(t) \) - position at time \( t \)
\( a(t) \) - acceleration

5.2.2 Environmental Forces

Given that the motions of a body are the result of forces, what forces should we try to model in our simulated environment? The forces relevant to ordinary objects under ordinary circumstances may be roughly divided into two classes, internal forces and external forces. Internal forces are those forces within a body which act to hold that body in one piece and help to maintain its shape. These forces tend to resist stretching and bending of the object. External forces act upon an object to affect its global motion through the environment. This would include such forces as gravity, air resistance, and contact. The following sections briefly describe actors designed to model the nature of these forces.

5.2.3 The Spring Actor

In order to maintain two mass actors at a fixed distance apart we connect them by an idealized spring. A spring actor perfectly obeys Hooke's law [Symon 1971];
the force exerted on the masses is linearly proportional to the difference between
the distance currently separating those masses and some equilibrium distance.
By applying equal but opposing forces along the line connecting the masses, the
conservation of momentum laws are upheld.

5.2.4 The Hinge Actor

This primitive is responsible for maintaining angular relationships within a config-
uration of mass actors. This can be used to maintain the shape of an object. For
example if four mass particles are arranged to represent two adjacent triangles, a
hinge may be used to maintain the angular relationship between the surface nor-
mals of the triangles. The hinge is assumed to follow an analogous statement of
Hooke's law for angular displacement.

5.2.5 The Aerodynamic Drag and Wind Actors

These actors allow the model to account for friction between the surface of a body
and the surrounding fluid. As in the hinge example, the drag actor assumes that
three mass elements represent a triangular section of a surface. The positions and
velocities of these masses are used to compute the position and velocity of the
triangular surface. The drag actor queries the wind for the density and velocity
of the air in the vicinity of the triangular surface, and computes an aerodynamic
drag force proportional to the velocity of the relative wind squared, the air density,
a drag coefficient (property of the triangular surface), and the cross sectional area
exposed to the relative wind (see [Kreider 1985], p.53).

5.2.6 The Gravity and Ground Actors

To simulate the force of gravitational attraction between two masses, the magnitude of the attraction would be determined by the law of gravitation. However, since the effects of gravitational attraction between mass particles are usually negligible, only the attraction between a particle and the ground are modeled. This is represented as a uniform acceleration field in the direction of the ground. The ground actor is represented as a flat plane.

5.2.7 The Contact Actor

Current software simulates elastic collisions between mass actors and a fixed ground plane. Collision is detected when the position of a mass actor passes through the ground plane. Collision is simulated by reflecting the perpendicular components of the velocity and position vectors across the ground plane. Inelastic collisions may be simulated by attenuating the magnitude of these reflected components. "Sticky" surfaces are simulated by "capturing" the mass and releasing it only when forces acting perpendicular to the ground plane are great enough to overcome the force of adhesion. To simulate surface friction, forces are applied tangential to the ground plane in opposition to the tangential velocity of the mass actor. This implementation of collision simulation is admittedly simplistic. However, the modular nature of the actor paradigm readily allows us to replace the current
collision detection model with a more sophisticated model such as the one presented by Hahn [Hahn 1988].

5.2.8 The Body Actor

The body actor represents a collection of actors at the group level; it represents an entire body as a subsystem of mass and force actors. The body actor simplifies the process of constructing models of physical objects. Since our current geometric modeling system uses polygonal surface representations, the body actor has been provided with the capability to construct a physical model from a geometric representation. The body takes triangulated polygonal descriptions and maps a mass actor to each vertex, a spring actor to each edge, and a hinge actor to each pair of adjacent triangles. In effect, this provides a bi-directional mapping between the physical and visual representations. The polygonal description is used to describe the topological arrangement of the physical model as well as provide a template for animating the positional information contained by the mass actors.

To control the physical properties of bodies from a global level, the body actor maintains a set of global properties for a group of actors. The body actor allows the user to modify the total mass of the body, the spring or hinge stiffness, or the drag factor, by simply sending a message to the body actor itself. The body actor then translates this message into the appropriate “sub-actor” messages and distributes these properties to the sub-actors as necessary. For example, if the
total mass is modified, the body actor will adjust the mass properties of each of
the mass sub-actors according to how the weight was distributed in the original
definition of the body.

The body actor also simplifies constructing relationships between groups of
actors; a body actor can “pass on” a relationship to the members of the group it
represents. For example, if a “flag” actor representing a group of “mass particle”
actors is to be related to the “wind”, the flag actor can generate the relationships
between each mass actor and the wind actor. This allows the user to deal only
with the simpler global level actors of “flag” and “wind”.

5.2.9 Animated Results

We have currently used behavioral simulation in two different ways. The first
method, termed forward simulation, is to simply set up a situation and let the
simulation run. This approach is useful for animating scenes in which the quality
of motion is more important visually than the exact path of the movement (for
example, a ball is to roll down a hill, but we do not really care where it goes). When
using forward simulation, the animator gives up direct control over the animation
(for example, the exact position over time of the ball). This technique is essential
when we wish to animate large collections of objects. In such cases, the animator
must rely upon the simulation to produce the correct quality of motion since it is
impossible for the animator to specify the motions of so many objects.
A second means of using simulation is to directly control some state variables of the environment using scripts, while allowing the remainder to be automatically controlled by the simulation. The most common use of scripts is to control the positions of some of the elements of a body (so called “dragging”). By applying scripts to the position property of a few mass actors in a flexible body, we can effectively “drag” the body about. Since the scripted mass actors are interconnected with the rest of the body’s mass actors via internal force actors, the body will be appropriately deformed.

More abstract animations can be produced by scripting the stiffness or equilibrium properties of a flexible object. For example, the shape of a body may also be changed by scripting the equilibrium angles of hinges or the equilibrium lengths of springs. By scripting the stiffness of a spring actor to suddenly diminish to zero, a breaking rubber band can be simulated. Finally, we can script external forces that act upon an object, either the naturally occurring forces such as wind and gravity, or invented “pseudo-forces” which can be used to push an object about.

To date, these simulation techniques have been applied to a variety of physical situations, which exhibit the general applicability of using simulation to generate animation. A demonstration of a bouncing flexible sphere, and a bouncing flexible rod show that realism in motion, collisions, and flexibility, are readily achieved [Haumann 1987]. The bouncing sphere displays the expected deformation of shape as it squashes upon contact with the floor and then rebounds (figures 5.1, 5.2, and
5.3). The rod, dropped at an angle to the floor, strikes at one end first, causing the rod to rotate about its center until the opposite end strikes. This shows that although the simulation models the rod locally as a collection of interconnected point masses, the global motion of the rod (the rotation) is reliably reproduced, and reflects the appropriate changes in angular momentum (figures 5.4, 5.5, 5.6, and 5.7).

A demonstration of a falling sheet of paper shows how interesting global motion of an object may be obtained by simulating the interactions of the physical properties of an object and its environment (see figures 5.8 and 5.9). In this demonstration the simulated sheet of paper is of low mass in comparison to the aerodynamic forces experienced by its surface. Although the paper does bend, it has been modeled to be stiffer than, say a piece of cloth. The driving force of gravity and the interplay between the aerodynamic forces and the shape of the paper as it flutters towards the ground, generates the unique shape of the path shown "stroboscopically" in figure 5.8. When the paper is moving in a direction normal to its surface, the aerodynamic force is large and rapidly retards any such motion. This causes the paper to pause and almost hover in place. Motion parallel to the plane of the paper causes almost no aerodynamic drag, hence these moments of hovering are usually followed by sideways accelerations as the paper slides off "downhill" in the direction (orientation) of least resistance. When an upturned corner "catches the breeze", the cycle repeats. Often in the process, the
Figure 5.1: Non-deformed falling sphere

Figure 5.2: Deformed sphere "squashing" on floor
Figure 5.3: Sphere rebounding

Figure 5.4: Non-deformed falling rod

Figure 5.5: Deformed rod striking the floor
Figure 5.6: Rod rebounding and rotating

Figure 5.7: Rod (far end now striking floor)
paper "flips" end over end. An example of this motion is shown in figure 5.9. The paper is rapidly descending edge first when the leading edge, slightly upturned, is caught by aerodynamic drag (figure 5.10). This causes a further upturning until the entire sheet has curved around and begins to move "uphill" (figures 5.11 and 5.12). At the top of the rise, the paper "hovers", then reverses direction, the once leading edge now becomes the trailing edge. When this cycle repeats itself, the paper again "hovers" and the opposite side of the paper now faces the ground. Thus, if the paper were colored red on one side and blue on the other, a viewer on the ground looking up might see first red, then blue, then red again.

At one time there was criticism of computer generated scenes because they appeared unrealistically "clean". It is interesting to note that "dirty" simulations are just as important to motion control as "dirty" renderings are to computer graphics. Experience has shown that clean simulations can generate a similar lack of realism. In the first attempt to animate the sheet of paper, it was modeled as a perfectly flat sheet, placed in the air at a moderate angle and released. The result was that the paper slid sideways, edge first, looking like a flat toboggan traveling down a perfectly smooth, invisible slope. It had been placed in (an unrealistically) perfect equilibrium situation and there was nothing to cause it to deviate from this configuration. To solve this problem the positions of the mass particles representing the surface were randomly perturbed small distances out from the plane of the surface, effectively wrinkling the paper.
Figure 5.8: Stroboscopic path of falling paper

Figure 5.9: Stroboscopic closeup of paper "flipping"
Figure 5.10: Paper beginning to slide “downhill”

Figure 5.11: Paper falling “downhill”

Figure 5.12: Paper beginning to move “uphill”
An even more interesting demonstration of the interplay between shape, gravity, and aerodynamic forces is portrayed by the animation of the flag in the wind. In this simulation, the wind is modeled as a vector field and controlled by a script which describes the wind speed and direction at each point in time. This script is the only user controlled variable modified during the animation; all the furling and flapping motion of the flag results from the simulation. Figure 5.13 shows the rippled shape, characteristic of a wind blown flag. In the animation, these ripples realistically form and travel across the surface of the flag (figures 5.14 and 5.15). In reality, these ripples are caused by pressure differences between the windward and leeward sides of the ripple. The simulation captures this action because the aerodynamic force on the windward side of the ripple forces it back into the plane of the flag while simultaneously pushing (from the opposite side of the flag) the leeward side of the ripple forward, out of the plane of the flag. The leeward side now becomes the windward side of the same ripple slightly displaced down wind. It is the combination of the aerodynamic forces and the internal forces holding the flag together in one piece, that cause the ripple to reform next itself, giving the appearance of the ripple "moving" down the length of the flag.

Figure 5.16 shows how we can even simulate "catastrophic" events in our simulated world. In this case we have caused the "clips" which hold the flag to the pole to "break" (this is accomplished by suddenly removing the constraints that require the masses at the two corners of the flag to remain fixed in space). The result is
Figure 5.13: Flag with ripple entering from left

Figure 5.14: Flag with ripple midway through

Figure 5.15: Flag with ripple moving off right side
as natural as one would expect; the flag blows away, pulled towards the ground by gravity. The flag realistically bunches up because the lengthwise tension, present while being held to the pole, has now been released.

Simulation has also been used to: animate a ribbon carried by a running robot [Girard 1986]; a curtain fluttering in the wind; to produce a wavy fluid effect in an expressionist film [Stavely 1988]; and the flexible cables on a dipping suspension bridge [Haumann 1987]. The curtain animation was similar to the flag animation except that the wind was modeled as a non-uniform field, being stronger at the window center than at the sides (figures 5.17, 5.18 and 5.19). The animation of the dipping bridge was a test portion of a film designed to teach children about the forces acting in a suspension bridge. The towers which support the overly flexible main cables first shrink, then rise back up. The mechanical simulation of the cables’ response displays a slinky-like quality.

The ribbon, in [Girard 1986] was simulated by essentially having the robot “drag” it about the environment. The location of the leading edge of the ribbon was determined entirely by the location of the robot’s hand. The rest of the flow and furling of the ribbon was due entirely to the interaction of aerodynamic drag with the inertial and stiffness properties of the ribbon’s surface.

One of the more novel uses has been to help create the goofy characters invented by Chris Wedge for his film Balloon Guy [Wedge 1987]. Figure 5.20 shows two such characters. The motion of the hair, ears, eyes, tongues, and attached strings
Figure 5.16: Flag released from flagpole

Figure 5.17: Static curtain shape

Figure 5.18: Curtain catching the breeze
Figure 5.19: Curtain settling back

Figure 5.20: Characters from the film Balloon Guy
were all controlled by a simulated response to the character's movements. The resulting animation is a comical display of wind blown hair, flopping ears, bobbing eyes, dangling tongues, and trailing strings. All of this motion, painful and nearly impossible to specify manually, was easily generated within a few hours. The only information that was needed to drive the simulation for each character was a script of position and orientation of the character's head for each frame of the animation. As each character spins or bobs, the attached hair, ears, and strings, realistically react to the character's motion. In effect, we are able to automatically reproduce the motion techniques demanded of the classical animators [Thomas 1981]: squash and stretch, follow through, and easing in and out.

Part of the fun of using simulation is the ability it gives one to play, to experiment with different physical properties of the object and to observe the resulting motion. During the making of Balloon Guy, we tried a variety of physical conditions. For example, increasing the mass of the hair would exaggerate the follow through of the hair, when the character stopped and started. Increasing gravity would cause all the appendages to droop more pronouncedly. Increasing the aerodynamic drag on the string would cause it to act more like a light thread, floating on the air. Decreasing the drag caused it to look more like a dangling chain being whipped about. All these experiments demonstrated to us that our single behavioral model of physical systems was capable of producing a wide variety of effects by simply changing a few physical parameters, and re-running the simulation.
5.3 Artificial Life - a Moth

The final test attempted to combine the aspects of a physically-based model with a model designed to simulate life-like behavior. This test utilized the aerodynamic drag actor model to simulate the wing motion of a simulated "moth-like" creature as it flapped its wings. The wing flapping was controlled by a script connected to the hinge actor aligned with the wing rotation axis. The aerodynamic drag of the wings against the surrounding air as the wings flapped caused the creature to flit into the air.

Although there was no feedback to adjust the movement of the wings, and hence this creature was not "goal-directed", the experiment is encouraging because it indicates the robustness of the behavioral simulation methodology. By building in realistic physical models at a low level of abstraction, more complex models may be constructed on top of them. In this way the high level models obtain the realism of the lower level for "free".
In at least one way the practice of art and science are similar: constrained by the limits of the tools at hand, creative solutions to current problems are sought. The differences between art and science, magnified by the societal view of them as separate disciplines, lie in their intended outcomes: emotion versus cold fact; and on the sometimes unfairly biased stereotypes placed upon them: superfluous versus necessary, humanistic versus abusive. By its very nature, this research is interdisciplinary, which makes it vulnerable to attack from both sides. Scientists may criticize the work as lacking rigor. Artists may criticize the results as confined and limited, perhaps inaccessible. Despite this, the main body of the research presented in this dissertation was carried out in an interdisciplinary environment where artists and scientists were in close contact, cooperation was encouraged, and joint projects were often launched and ushered to fruitful conclusions. The results of this intermingling were healthy cross-pollination of ideas, sharing of frustrations, and realization of common goals. Needless to say, the tension between
these similarities and contrasts present in this environment provided the driving energy for much of the work presented here.

6.1 Contributions

Behavioral simulation, as defined and used in this research, is a proven means to increase the temporal complexity in computer generated scenes. The ability to generate complex motion from simple rules of behavior governing the interactions of large collections of moving elements is a new and exciting way to achieve complexity through terse input. The emergence of global phenomena from locally occurring interactions models closely the way complexity appears in nature, and can result in convincingly realistic movement.

The main contribution of this research has been to present an architecture for the construction of a software test-bed in which experimental behavioral simulations may be developed and tested. The utility of such a test-bed has proven useful in the domain of computer animation wherein complex physically-based models are required to achieve realistic visual effects. The success of these experiments supports the behavioral simulation approach for the following reasons. The very nature of the reductionistic approach to complex systems has the advantage of intuitive clarity, particularly for non-technical animators that might need to utilize such techniques. Such models mesh well with the animators mental view of the worlds being simulated. In addition, the object-oriented approach readily
supports the combination of behaviors by layering one on top of the other. In this way higher levels of abstraction may be constructed to model more complex systems, while still retaining the desired constraints provided by the lower levels. In this manner, behavioral simulation allows artists to readily cast their models into forms at an appropriate level of abstraction that is easy for an artist to manipulate, yet the accessibility to the features of the lower levels of abstraction can be retained.

It must be pointed out however, that behavioral simulations are usually non-linear and possibly chaotic. This means that small changes in the initial conditions that begin a simulation or small changes in a script that controls the animation may result in vastly different system behaviors. In addition, the parameters which are used to control a simulation are not constrained to act independently in their global effects upon the behavior of the system. These characteristics may make some models difficult to control, and this may limit the utility of using forward simulation in animation until the behavior of such systems is better understood. ¹

In order to deal with this problem a technique of scripting is presented which allows the user to steer the course of the simulation by explicitly controlling some system properties, while still obtaining the benefits of procedurally generated motion.

¹On the other hand, such unpredictability may be used to an artist's advantage. Part of being creative is dealing with unexpected outcomes that one encounters in the medium in which one is working.
Another disadvantage of using a discrete structured approach is the computational expense of simulating the behaviors of all the elements within an environment. This places a fundamental limit upon the complexity of the simulation. However, because behavioral simulation is an experimental methodology which mirrors the concurrency inherent in the natural system being modeled, it can readily take advantage of parallel processing techniques.

6.2 Further Research

The development of the behavioral test-bed paves the way for constructing behavioral simulations of many different types of phenomena in addition to the physically-based modeling and the cognitive character models hinted at by this research. Clearly there is much to be done in the physically-based realm alone. A more sophisticated collision detection model is needed if greater realism is to be achieved. A wider variety of physical object types such as those suggested in [Terzopoulos 1988b] need further exploration. Fluid models of the type suggested by the work in lattice gas automata deserve further inspection.

The area of cognitive modeling is a fertile area for research. Collision avoidance models augmenting those suggested in [Reynolds 1987] might result in rich possibilities of interesting animations of group behaviors. Coupling such models with simulations of information flow within groups might yield useful results. Consider modeling the dynamics of an ant colony or beehive, and how the location of food
is reported back to the nest. Such a model might even prove useful for studies in pest control or honey production.

The use of simulation techniques in general will become increasingly prevalent in animation systems of the future. Only in this way can motion generation be automated to the extent that complex, interesting scenes of realistic motion can be mass produced. By looking at ways in which simulation can be adapted to the needs of the animator, we will be better able to harness this power to meet the creative demands of tomorrow's world.
Appendix A

Foundations of Physically-Based Modeling

This appendix is intended to serve as a brief review of topics relevant to techniques used in physically-based modeling.

A.1 Classical Mechanics

Classical mechanics is the study of the motion of objects. Dynamics is concerned with how forces cause objects to move. Issac Newton's laws of motion form the foundation for classical mechanics. His first law, the law of inertia, tells us that the momentum of an object (the product of its mass and velocity) remains unchanged if no force acts upon the object.

A few points must be illuminated here. First of all, the velocity of an object is a vector quantity having a length (representing the speed) and direction. Mass is a scalar indicative of the amount of matter contained in the object. The momentum is therefore also a vector quantity. If an object is assumed to have a constant mass, we may then interpret this law to mean that in the absence of any forces, the
velocity of the object will not change. Note that this implies a constant direction as well as speed! An object following a curved path has a changing velocity and hence there must be a force acting upon this object.

Another idea is that in most ordinary situations, friction forces are always present. The frictional forces of aerodynamic drag for example are responsible for bringing thrown objects eventually to rest. They are also responsible for making self propelled objects such as rockets flying in the atmosphere travel in uniform (constant velocity) motion in spite of the fact that the rocket engine is exerting a force on the rocket. This often misleads people to the naive view that a force (the rocket engine) is needed just to maintain uniform motion (i.e. forces maintain velocities not accelerations, which is not true). In fact what is really happening is that the forces of friction are exactly cancelling out the driving force of the rocket engine, hence no acceleration occurs, since there is no net force.

Newton's second law mathematically states how a body behaves in the presence of a force, namely:

\[ F = Ma \]  \hspace{1cm} (A.1)

where the net vector force \( F \) acting on the object is equal to the product of the mass \( M \) of the object times the acceleration \( a \) experienced by that object. This mathematical statement tells us exactly how the motion of an object will be affected when forces are applied to it.
Newton's third law states how forces only arise in pairs and as the result of the interactions between objects. This is the action/reaction law. If object A applies a force \( F \) upon object B, then object B applies an equal but opposite force \( -F \) upon object A. This law is essential when we desire to reproduce the realistic motion of objects which are interacting (collisions, for example).

There are two corollaries to this last action/reaction law. They basically state that for a collection of masses, treated as a single system, all forces which arise from the interaction between them cannot result in a net increase in the energy of that system. The energy of a system is the capacity of the system to perform work. This idea leads directly to the conservation of linear and angular momentum laws for systems of particles.

**A.2 Hamilton's Principle and the Lagrange Equations**

It is common practice in the physical and mechanical sciences to manipulate the equations of motion in the Lagrangian form rather than the Newtonian form. The reasoning for this is that the Newtonian form becomes unwieldy (or impossible to use) in problems where the motion of the object is constrained in some manner. Consider the problem of describing the motion of a bead as it slides downhill along a curved wire. Since the bead remains attached to the wire, there must be certain forces of "constraint" which impose this condition. At any instant in time these forces will depend upon the local shape of the wire and the current velocity of the
bead. In general it is difficult to explicitly represent these forces in the Newtonian form which includes a term for the TOTAL (net) external force acting on the object over time. By implicitly using Hamilton's principle (described below) the Lagrangian form of the equations may be derived and used to circumvent some of the practical difficulties of directly applying Newton's equations.

A.3 Work and Energy

The work \(W\) done upon an object is defined to be the integral of force \(F\) with respect to the distance \(x\) over which it acts.

\[
W = \int(x_1 - x_2)F\,dx
\]  \hspace{1cm} (A.2)

Note that work is a scalar. Work done on a system or by a system represents a transfer of energy between objects or a change in the form of that energy.

Energy may take two forms: potential energy and kinetic energy. The kinetic energy of an object represents the ability of an object to perform work based upon its motion. The kinetic energy \(T\) of an object depends upon its mass \(m\) and its velocity \(v\) as represented by the equation:

\[
T = \frac{1}{2}mv^2
\]  \hspace{1cm} (A.3)

A bowling ball rolling down an alley has quite a bit of kinetic energy. When it hits the stationary pins at the end of the alley, some of that energy is transferred from the ball to the pins. The ball having lost some of its energy to the pins, slows
down. Those pins struck by the ball (hopefully all ten!) have gained energy as evidenced by their subsequent motion.

The potential energy of a system is the energy "stored" by a system and represents the "potential" of that stored system to perform work. Such stored energy is represented by a stretched spring or a weight suspended in a gravitational field. Both have potential for performing work; the suspended weight can do work as it falls, the spring can do work as it relaxes. Indeed, timepieces of old used both of these methods to store the energy needed to perform the work of keeping time. The battery which has replaced both of these methods in modern timepieces is simply a chemical means to store "potential" energy.

Measuring the potential energy of a system depends upon how the energy is stored. For example, if a weight is lifted $x$ feet higher than some rest position, then the stored potential energy in that lifted weight is equal to the work done in lifting it:

$$U = \int mg \, dx = mgx$$  \hfill (A.4)

(where $m$ is the mass of the object and $g$ is the acceleration of gravity.) The energy in a system may be converted from one form to another. Consider a system consisting of a spring fixed at one end and attached to a mass at the other. Assume that the system is frictionless and that the mass is free to move. If one grasps the mass, pulls it so as to stretch the spring, and holds it in that stretched configuration, one has done work on the system and added energy to the system.
in the form of potential energy stored in the stretched spring. A stretched but motionless spring is an example of such a stored potential for work. If the mass is now released, the spring transfers its potential energy into the kinetic energy of the mass. The spring loses energy as it returns to a relaxed state, whereas the mass gains energy as its velocity increases. This concept of energy changing forms is important to animation because it again captures how objects' motions may change as the result of interactions between them.

Total energy of a system is the sum of the potential and kinetic energies. Note that in a conservative system this total conforms to the conservation of energy law; this sum remains a constant. Thus in the spring/mass system the kinetic energy of the mass plus the potential energy stored in the spring is some constant.

A.4 Hamilton's Principle

Hamilton's Principle is a minimizational principle that simply states that:

"Of all the possible paths along which a dynamical system may move from one point to another within a specified time interval (consistent with any constraints), the actual path followed is that which minimizes the time integral of the difference between the kinetic and potential energies."

([Marion 1970] pp. 198)
In other words, we wish to find the extremum of the following integral:

\[ I = \int (T - U) \, dt \]  \hspace{1cm} (A.5)

Normally, if we are given an ordinary equation we can solve for extreme values by setting the first derivative to zero. However, Hamilton's principle is stated in terms of an integral equation. The value of this integral depends upon the function inside the integrand, which takes the form \( F(t, x(t), v(t)) \), i.e. it is not a simple variable. The value of the integral is a function of the integrand (a function). Hence it is termed a functional. In order to differentiate a functional we are required to use the calculus of variations. For full treatment of this subject see [Gelfand 1963] or [Weinstock 1952].
Bibliography


Haumann D., Caldwell, C., Foss, G., “Dynamic Simulations of Flexible Objects”, A film first shown at the Siggraph '87 Film and Video Show, Anaheim, California, July 1987.


