A Multi-Threaded Simulator for the Kinetics of Virus Shell Assembly

by

Russell S. Schwartz

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 10, 1996

Copyright 1996 Russell S. Schwartz. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author ______________________

Department of Electrical Engineering and Computer Science
May 10, 1996

Certified by ______________________

Bonnie Berger
Assistant Professor of Applied Mathematics
Thesis Supervisor

Accepted by ______________________

F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 11 1996

Eng.

LIBRARIES

A Multi-Threaded Simulator for the Kinetics of Virus Shell Assembly

by

Russell S. Schwartz

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

How icosahedral virus shells form has been a long-standing open question in Virology. One approach to answering this question, the local rules theory of virus shell assembly, has provided significant insights. However, prior work with this theory and simulations based on it have been very abstract; they have, therefore, been unable to answer some important questions. To address this, a graphical simulator providing a more realistic physical model of viral coat protein interactions has been developed. This simulator has several features that make it both powerful and easy to use, including a versatile model of the underlying physical systems, a high-level control language, a graphical user interface, and a multi-threaded design that allows the use of parallel architectures. Preliminary work with the simulator suggests that it will be a valuable tool for understanding icosahedral virus shell assembly.

Thesis Supervisor: Bonnie Berger

Title: Assistant Professor of Applied Mathematics

Acknowledgments

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

I am especially indebted to my thesis supervisor, Bonnie Berger, for her guidance and support throughout this project, for giving me the freedom to pursue it, and for being patient and understanding when various setbacks came up. I would also like to thank Peter Prevelige for his suggestions on the design of the project and for helping me keep it connected to the real world. I am likewise grateful to Peter Shor for his advice on solving some difficult problems. Thanks also go to Bobby Blumofe for getting me started with Cilk and patiently answering my questions. Finally, I would like to acknowledge my family for their continuing support and encouragement.

Contents

1 Introduction **8**

2 Background and Motivation **10**

2.1 Icosahedral Virus Shell Assembly 10

2.2 Local Rules 12

2.3 Empirical Support for Local Rules 13

2.4 Prior Simulation Work 15

2.5 Conclusions and Motivation 19

3 Design Requirements **21**

3.1 Functionality 21

3.2 Performance 23

3.3 Other Constraints 24

4 Implementation **25**

4.1 The Physical Model 25

4.1.1 Coat Proteins 26

4.1.2 Binding Interactions 30

4.1.3 The Environment 33

4.2 The User Interface 34

4.2.1 The Command Interpreter 34

4.2.2 The Graphical Interface 35

4.2.3 The Graphics Display 39

4.2.4 The Controller 41

4.2.5 The Alternate Controller . . . 42
4.2.6 Data Files . . . 43
4.3 The Serial/Parallel Interface . . . 47
4.4 Major Algorithms and Numerical Methods . . . 48
4.4.1 Approximation of the ODE . . . 48
4.4.2 Advancing Time . . . 50
4.4.3 Temperature/Brownian Motion . . . 55

5 Evaluation **58**
5.1 Functionality . . . 58
5.2 Performance . . . 60
5.3 Other Constraints . . . 62

6 Applications **63**
6.1 Undirected Assembly . . . 63
6.2 Modeling with the Alternate Controller . . . 65

7 Discussion **67**
7.1 Conclusions . . . 67
7.2 Future Work . . . 69
7.2.1 Potential Improvements to the Simulator . . . 69
7.2.2 Other Applications . . . 70

A Specifications for the Control Language **75**
A.1 General Features . . . 75
A.2 Hooks to the Graphical Interface . . . 77
A.3 Primitive Routines and Special Forms . . . 81
A.3.1 Special Forms . . . 82
A.3.2 General Operations . . . 83
A.3.3 Graphics Routines . . . 86
A.3.4 Communication Routines . . . 87
A.3.5 Vector Routines . . . 89

List of Figures

2-1 T=1 Local Rule 13
2-2 T=1 Lattice 13
2-3 T=1 Shell 14
2-4 T=7 Local Rules 15
2-5 T=7 Shell 16
2-6 Alternate T=7 Local Rules 17
2-7 Polyoma Virus Rules 17

4-1 Single Node 27
4-2 Complex Node 27
4-3 Variable Nodes 28
4-4 Complex Node with Variable Region 29
4-5 Bond Forces 32
4-6 Sample Controller Code 35
4-7 Graphical User Interface 36
4-8 Sample Workspace 40
4-9 Rules File 44
4-10 Save File 45
4-11 Control File 46
4-12 Domain Decomposition 52
4-13 Poor Decompositions 53
4-14 Pseudo-code for Advancing Time 54

5-1 Performance Tests 60

6-1 Undirected Assembly 64
6-2 Simulated Shells 66

Chapter 1

Introduction

This thesis paper describes the development of a simulator for studying virus shell assembly. The simulator is meant to provide realistic data, in both numerical and graphical formats, on the assembly of certain kinds of virus shells; to this end, it incorporates physically reasonable models of some of the kinetic and thermodynamic processes involved in virus shell assembly. The thesis is part of an ongoing project, led by Prof. Bonnie Berger, to study the mathematical basis of virus shell assembly.

Although the thesis project is the development of the simulator itself, the thesis paper presents some additional material in order to aid in understanding the project. Because of the interdisciplinary nature of the work, it is helpful to provide some background on the problem. The paper describes the design constraints on the simulator as a means of showing the motivation behind many of the features. In addition, it shows two sample applications. These give a general demonstration of the capabilities of the simulator, rather than an exhaustive overview of its functionality.

The thesis is organized into seven chapters. Chapter 1 contains a general statement of the problem and this overview. Chapter 2 describes in more detail the problem the thesis addresses, prior work done in the area, and how the thesis project fits into the overall research effort. Chapter 3 discusses constraints on the design of the simulator. Chapter 4 examines how the simulator works in detail, describing both the functionality of the simulator and the underlying mechanisms by which it works, and explaining the important design decisions. Chapter 5 evaluates how well the simula-

tor meets its design goals in practice. Chapter 6 describes two example applications. Finally, Chapter 7 draws some conclusions about the project and outlines some potential future work involving the simulator. Appendix A provides specifications for the simulator's control language.

Chapter 2

Background and Motivation

This chapter describes the problem of virus shell assembly and how it motivates the construction of a new simulator. Section 2.1 describes some major issues in understanding icosahedral virus shell assembly. Section 2.2 discusses the local rules theory of virus shell assembly. Section 2.3 describes some of the empirical support for local rules. Section 2.4 describes prior simulation work on this project. Section 2.5 draws some conclusions about the prior work and describes how it motivates the thesis project.

2.1 Icosahedral Virus Shell Assembly

Determining how viruses grow in a cell is a crucial problem to understanding their life-cycle and how it might be disrupted. In its most basic form, a virus consists of genetic material surrounded by a protein coat, although there are often other components. A critical problem in understanding virus growth is finding how the protein coats, or shells, of viruses form. Viral coats are typically made of several hundred copies of a single protein that assemble into a closed shell around the genetic material. These coat proteins are often capable of self-assembling into a completed shell, although sometimes additional proteins, called scaffolding proteins, or interactions with the genetic material are required[12, 6]. The process by which self-assembly occurs is not fully understood. There have been attempts to explain the process, some of which

appear quite successful for helical viruses, which are characterized by long, cylindrical coats. However, these theories have been less successful in explaining icosahedral viruses, those whose shells exhibit icosahedral symmetry.

Icosahedral viruses are an important subclass of viruses including many animal, plant, and bacterial viruses, as well as almost all human viruses[11]. Human diseases caused by icosahedral viruses include the common cold, chicken pox, polio, herpes, and yellow fever[22]. Icosahedral viruses are defined by two-fold, three-fold, and five-fold symmetries between the protein subunits in the shell. The result is a shell consisting of twenty identical faces, each three-way symmetric. Icosahedral viruses have traditionally been described in terms of capsomers, i.e. hexagons and pentagons of coat proteins which are called hexamers and pentamers, respectively. Different sizes of shells are classified according to triangulation, or T, numbers by the relative positions of the hexamers and pentamers. A complete virus shell of T number n contains $60n$ coat proteins. Although these coat proteins are often all chemically identical, they can occupy geometrically different binding environments when the T number is greater than one. This fact has been the source of much of the difficulty in understanding icosahedral virus shell assembly.

The traditional explanation for virus shell assembly is the quasi-equivalence theory of Caspar and Klug[7]. According to this theory, coat proteins occupy a single basic shape, but have a limited amount of elasticity which allows them to be deformed slightly from that shape as a result of their overall placement in the shell. Caspar and Klug hypothesized that this elasticity allows the proteins to occupy non-equivalent but similar, or "quasi-equivalent," binding environments. Based on this hypothesis, Caspar and Klug determined that the different binding environments would be sufficiently similar to fit the quasi-equivalence theory only if proteins occur in certain patterns of pentameric and hexameric binding environments. From this restriction, they concluded that only T numbers of the form p^2+pq+q^2, for non-negative integers p and q, should be possible. Caspar and Klug hypothesized that in the self-assembly of virus shells, coat proteins first assemble into hexamers and pentamers; these then come together to form a complete shell.

2.2 Local Rules

Another explanation for how icosahedral virus shells form is the local rules theory of Berger *et al.*[3]. The local rules theory proposed that coat proteins do not require any high-level knowledge of their position in a completed shell, only local knowledge about their neighbors in the shell. Under the local rules theory, a given coat protein can take on distinct shapes, or conformations. These conformations follow sets of local rules, where a local rule specifies the conformations of the neighbors of a given protein and their relative positions in the shell. A set of local rules can uniquely define the geometry of a given icosahedral shell.

The simplest icosahedral shell, a T=1 shell, requires only a single local rule. This rule is shown in Figure 2-1. The partial circles specify that a coat protein has three neighbors, all of conformation 1. The angles between the three neighbors, going around the protein, are 120°, 108°, and 120°, as specified in the rule. These do not add up to 360° because they do not lie in a plane. Attaching several nodes obeying these rules gives the partial lattice shown in Figure 2-2, which is composed of regularly spaced hexagons and pentagons. A computer simulation repeatedly applying the T=1 rule produces the closed shell shown in Figure 2-3.

A more interesting example of local rules is the T=7 shell. A set of rules for this shell is shown in Figure 2-4. Because this set of rules has seven distinct conformations, proteins must know not only the relative positions of their neighbors, but also their conformations. Computer simulations repeatedly applying this set of rules produce the closed T=7 shell shown in Figure 2-5.

Although there is evidence for the correctness of local rules, there is some reason to believe that the previous set of local rules may not be correct for at least some actual T=7 viruses. Berger *et al.*[3, 2] proposed an alternative system of rules for the T=7 shell that appears more consistent with the biological data. These alternative rules use only four distinct conformations, by hypothesizing that hexamers exhibit 180° symmetry and therefore require only three conformations. This set of rules, however, allows for more than one rule per conformation. This creates an ambiguity in the

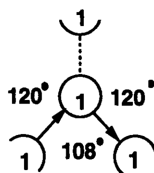


Figure 2-1: A Local Rule for a T=1 Virus Shell

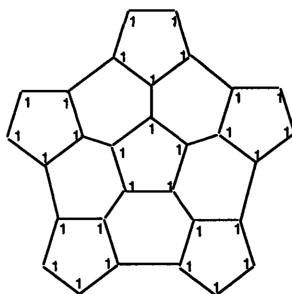


Figure 2-2: A Partial Lattice of a T=1 Virus Shell

assembly process, which is resolved by disallowing a particular configuration of nodes and adding the additional constraint that assembly must begin on a pentameric node, that is a node of type 1. This alternate set of rules is shown in Figure 2-6. Following this set of rules, with the specified constraints, uniquely defines a shell geometrically identical to the previous T=7 shell, but containing only four conformations.

2.3 Empirical Support for Local Rules

There is considerable empirical support for the local rules. Rossmann[21] noted that elastic deformations cannot account for structural changes in coat proteins observed experimentally, suggesting that coat proteins do shift between distinct conformations. Other evidence has supported this idea of conformational shifting[13, 15, 18]. While the existence of such conformations is problematic for quasi-equivalence, it is completely compatible with local rules. The local rules theory can also explain some viruses that do not fit into patterns predicted by quasi-equivalence. For example, Figure 2-7 provides a set of local rules for the binding pattern of polyoma virus, which does not conform to the lattice patterns considered viable under quasi-

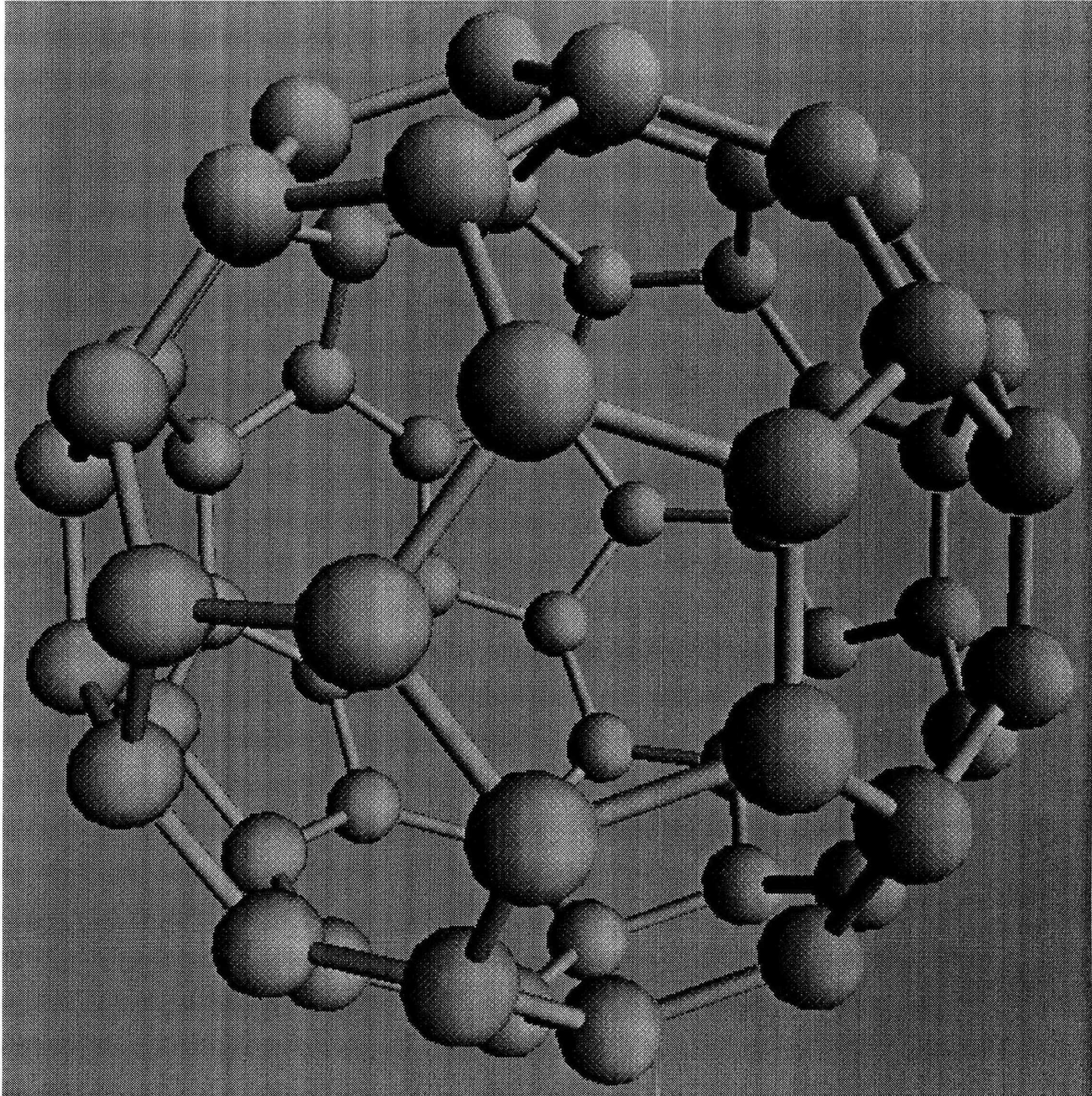


Figure 2-3: A Simulated T=1 Virus Shell

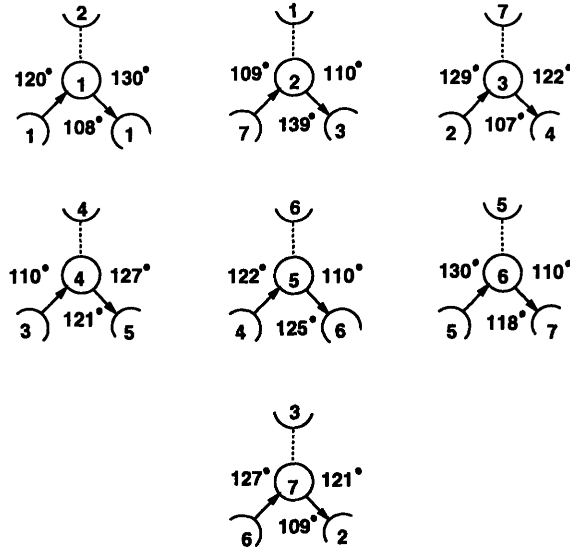


Figure 2-4: Local Rules for a T=7 Virus Shell

equivalence[19, 1].

Empirical support for local rules is particularly strong for the alternate T=7 rules described above. The hypothesized symmetry of hexamers is visible in micrographs of the T=7 bacteriophage P22[18, 23]. Further, the rules suggest a relationship between T=4 and T=7 shells; this relationship does exist in nature. For example, the bacteriophage P2 has a satellite phage P4 that constructs its shell from P2's coat protein. However, while P2 is a T=7 virus, P4 is a T=4 virus[8]. Earnshaw and King[9] further found that P22, when grown in the absence of a scaffolding protein, occasionally produces T=4 shells. Katsura *et al.*[14] found that a mutation in the coat protein of the T=7 phage lambda could produce functional T=4 shells. Based on the alternative rules, Berger and Shor predicted possible locations of scaffolding protein for T=4 and T=7 shells that are consistent with data since found for P4[16] and P22[23].

2.4 Prior Simulation Work

Previous work on this project involved the development of an earlier simulator for local rules[17]. This simulator represented coat proteins abstractly as spherical masses

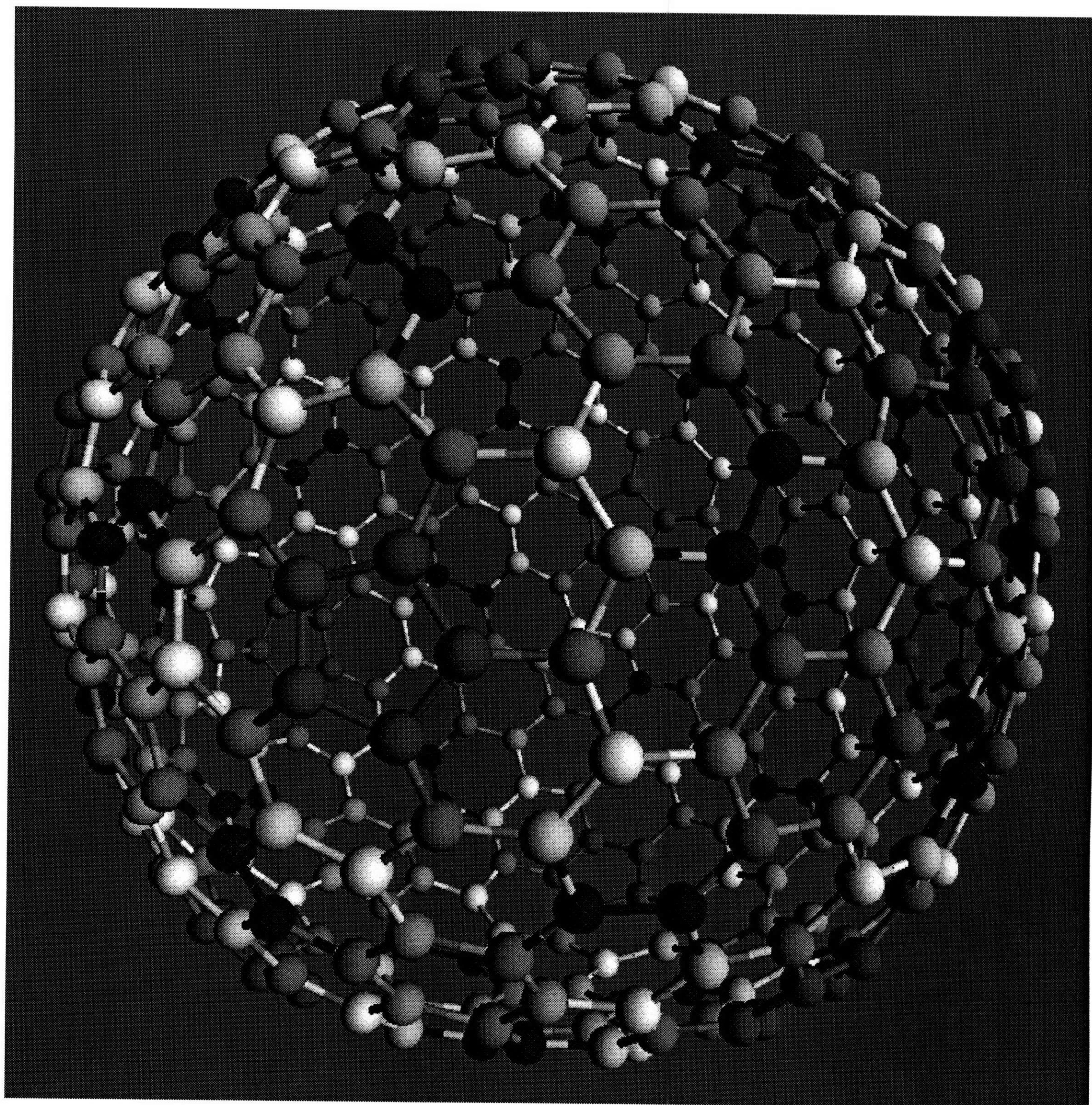


Figure 2-5: A Simulated T=7 Virus Shell

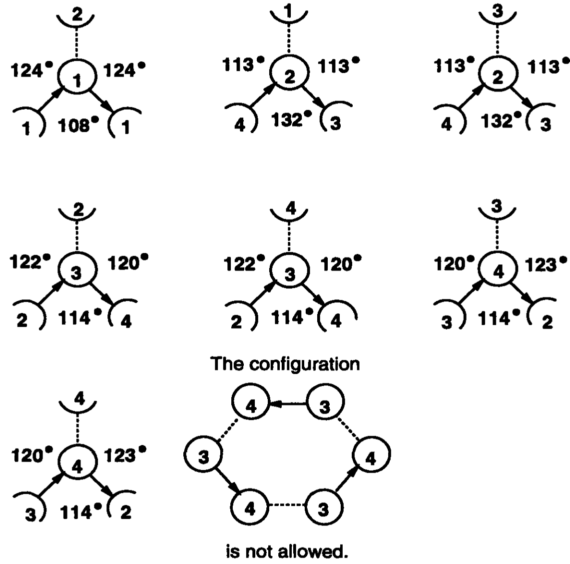


Figure 2-6: Alternate Local Rules for a T=7 Virus Shell

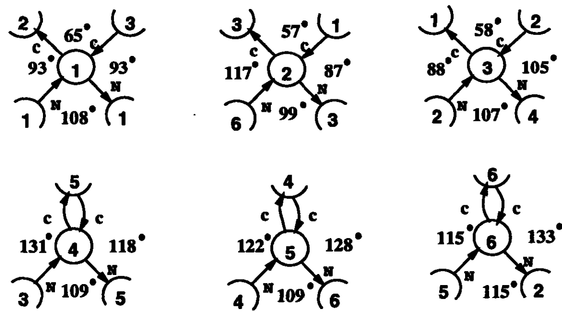


Figure 2-7: Local Rules for Polyoma Virus

connected by springs. With this simulator, a user could define a set of rules, such as those described above, which could generate a shell. The simulator would begin with a single protein and add new proteins one at a time, relaxing the stresses in the shell after each addition until the energy of the inter-protein bonds dropped below a fixed tolerance.

This earlier simulator was applied to several tasks. In addition to creating the simulated T=1 and T=7 shells shown earlier, it produced shells of a variety of different sizes and configurations. It also demonstrated the versatility of the local rules; for example, it could generate the non-quasi-equivalent polyoma virus shell described in Section 2.3. The simulator could further demonstrate the robustness of the local rules by testing the tolerance of rules to random perturbations. Another important application of this simulator was to study shell malformations. Berger *et al.*[3] found that replacing a single pentamer with a hexamer at the start of shell growth would produce a spiraling malformation, similar to malformations observed in nature. The simulator was also used to explore deliberately inducing malformations. Introducing a "poisoned subunit," which attaches to a shell but does not follow the local rules, into a simulation resulted in a shell malformation, suggesting a strategy for interfering with shell growth in practice. Finally, there has been limited work with the earlier simulator in modeling alternate rule sets such as the four conformation T=7 rules described in Section 2.2.

Despite this simulator's successes, there were limitations to what it could do. The simulator could model local rules but could not provide information on how they might be physically implemented, such as how important the shape of a subunit is, how subunits bind to each other, or how strong inter-subunit binding interactions must be. The simulator could not provide some potentially important quantitative data, such as that concerning reaction rates and pathways. Such information could be significant in finding likely avenues for attacking shell growth. Moreover, while the simulator could suggest strategies for disrupting shell growth, it could not provide specifics; it could indicate that poisoned subunits might be successful in attacking a shell, but could not say what physical properties such a subunit would need, or what concentration would

be required in a cell in order to reliably inhibit shell growth. These are important questions in evaluating the biological feasibility of such techniques. The simulator also could not adequately model the ability of proteins to detach after attaching to a shell, or the effect of different models of binding kinetics on the order of assembly. Finally, it could not realistically model conformational switching, which appears to be crucial to understanding and simulating shell growth; this is particularly problematic in modeling alternate rule sets, for which conformational switching is proposed to occur after a protein has attached to the shell. It may be difficult or impossible to obtain any of this data experimentally.

2.5 Conclusions and Motivation

The aforementioned limitations could seriously impede the progress of the broader research effort. Previous work on the project has depended heavily on simulation results, both as a verification of concept and as a substitute for difficult to obtain experimental data. It is likely that simulation results could be similarly valuable in continuing work. This suggests that it would be useful to find some way to simulate a broader range of phenomena.

One possible approach to this problem would be to attempt to extend the earlier simulator. This has in fact been done several times, to allow some work with alternate rules and to allow minimal modeling of kinetics. However, the required data is largely incompatible with the physical model used by the prior simulator. That simulator, because of its emphasis on energy relaxation rather than advancing quantifiable time, would require substantial modifications before it would be suitable for this application. Furthermore, it cannot easily be extended to handle free-floating proteins, which is necessary for a valid model of kinetics. Therefore, the prior simulator is not suitable as a base for beginning such work.

It thus appears that a new simulator, capable of addressing the limitations described above, is required. Such a simulator would have to adopt a substantially less abstract model of shell assembly than the prior simulator. It would require more

realistic models of quantifiable time, coat proteins, and the kinetics of their interactions. The new simulator could then be used as a supplement to the earlier simulator to address quantitative questions that are not easily examined in the laboratory and cannot be handled by the prior simulator alone.

Chapter 3

Design Requirements

A necessary step in constructing and evaluating a new simulator is deciding what characteristics it must have in order to fulfill its purpose in the broader project. This chapter discusses the design requirements necessary to allow the simulator to perform its intended tasks. Section 3.1 describes the functionality required of the program. Section 3.2 examines performance constraints. Finally, Section 3.3 discusses some additional constraints that must be placed on the simulator design.

3.1 Functionality

In order to meet the needs of the project, the new simulator requires a more realistic model of time and space than the prior simulator used. Rather than modeling a single forming shell, the simulator should focus on a "soup" of free-floating particles capable of interacting with each other. Rather than allowing proteins to add one at a time without regard for the interval between additions and with the order of additions assumed, proteins should be free to assemble in any order and at any rate. This freedom is essential to gathering quantitative data on the assembly process. Energy relaxation, used in the previous simulator after each subunit addition, is not meaningful when using free-floating particles; the new simulator must therefore maintain a quantitative model of elapsed time.

Also important in gathering quantitative data is a more realistic model of reaction

kinetics as it relates to binding interactions. The simulator should support a model of binding that allows for bonds to form and break probabilistically, where the term "bond" is used loosely hereafter to refer to the binding interactions between two subunits. Furthermore, the probabilities of these events should be based on a physically valid model of kinetics, adjustable according to user-supplied binding energies. This would allow a user to test the effects of different thermodynamic models on assembly rates and pathways, and allow easy integration of laboratory results into the simulator model.

Another important consideration is a more physically reasonable model of coat proteins. It should be possible to design simulated proteins with different shapes and bond configurations. Furthermore, physical properties of proteins, such as mass and size, should be user-definable using physically realistic values. This should allow experimentation with physical implementations of local rules and testing of how modifying different physical properties affects shell assembly. The simulator should also support conformational switching. This means that it should allow subunits to change shape or bond configuration probabilistically during a simulation. As with binding, the probabilities should be configurable by users and tied to a realistic model of kinetics. This would likewise aid users in testing the effects of different thermodynamic models on a simulation.

The simulator must also be configurable to a range of demands. In part, this means that many of the parameters that would be reasonably expected to vary between simulations, such as types of particles or temperature of the solution, should be configurable without modifying the code. It also means that the simulator should be adaptable to changes in what a user might require of it. There is a trade-off involved here; trying to anticipate every possible use would result in a simulator that is large, slow, and prone to bugs. However, the simulator should have sufficient versatility that another simulator will not be required to investigate questions that can reasonably be anticipated in the broader research effort.

The final design constraint is that the simulator be reasonably easy to use. It is meant to be a tool not just for computer professionals, but also for biologists who

may be less experienced with the use of computers. Less experienced users should be able to run a simulation, modify simple parameters, and examine the results without much specialized knowledge of the simulator. As with configurability, there is some trade-off involved here, as ease of use must be weighed against versatility. Therefore, it is acceptable to make less frequent or more specialized operations require more knowledge, provided the most common and important options are easily accessible.

3.2 Performance

Another important design consideration is how quickly the code will run. Although there is no hard upper limit on how long the code can take to solve realistic problems, in order for the simulator to be practical it must be possible to generate a virus shell in a reasonable time. The run-time of similar molecular dynamics simulators is generally measured in tens of hours. Since a virus shell typically contains several hundred protein subunits, it should be possible to simulate on the order of one thousand subunits for enough simulated time to grow a shell on a similar time scale.

Furthermore, this performance must not come at the expense of too much accuracy. In simulation problems there is often a trade-off between the amount of work required and the precision of the solutions generated. In order to generate reliable data, it is necessary that the simulator use work-efficient numerical methods, allowing sufficient accuracy without compromising performance. The meaning of accuracy in this context must be precisely specified, as roundoff errors make it impossible to determine particle positions and trajectories to any fixed degree of accuracy over an arbitrary length of time. In this case, there are two kinds of accuracy required. The first is a local accuracy, meaning that over a short time, a particle should follow the path predicted by Newtonian dynamics given its starting state and the forces acting on it. Thus, for example, particles will not pass through each other, and those bonded together will tend to stay close together spatially. The second kind of accuracy is a long-term statistical accuracy. This means that particles should, even over arbitrarily long times, consistently adhere to the rules of statistical thermodynamics, by, for

example, conserving energy.

Finally, certain potential bottlenecks should be avoided in order to make the simulator usable. The simulator is meant to be interactive, and therefore the graphics cannot be a major bottleneck. Otherwise, accessing the simulator would become frustrating for users. Furthermore, shared resources, particularly memory and network bandwidth, must not be overly taxed, or else the simulator could slow down intolerably in high-use environments or could become a nuisance to users of other programs.

3.3 Other Constraints

One additional constraint required to make the project both useful and feasible is that it must be possible to run the simulator on available hardware. At the present time, the main hardware available for the project is a variety of Unix workstations. This indicates that the simulator should be accessible through such machines. However, particular workstations or types of workstations are often unavailable, suggesting that the simulator should be portable to different Unix platforms. In addition, a Connection Machine CM-5 parallel computer is available. While it is reasonable to take advantage of this resource, the simulator should also be usable without access to the CM-5.

A final constraint is that the scope of the project should be sufficiently limited that it is possible to develop a working simulator in about a year. This is necessary because the simulator is one component of an active research project, and the project as a whole cannot be delayed on account of it. This development time must consist not only of coding, but of all other phases of software construction. This includes any necessary research into the underlying physics and mathematics, the design and planning of the simulator, coding, testing and debugging, and documenting the project. There should additionally be some margin for error, as it is impossible to precisely predict the time required for these steps.

Chapter 4

Implementation

This chapter describes the design of the simulator in detail, explaining how specific aspects of the design are meant to fulfill the design goals. From the user's point of view, the simulator represents a way of interacting with a mathematical model of the relevant physical systems. The distinction here between the mathematical model and user interactions with it is not just an abstraction; the code is split into two parts corresponding to this abstraction. The first of these is the numerical simulator itself, a multi-threaded program designed for use on a parallel supercomputer. The second part is a user interface meant to be run on a workstation. The remainder of this chapter describes these two parts and how they interact. Section 4.1 describes the physical model the simulator uses. Section 4.2 discusses the interface through which the user interacts with the underlying model. Section 4.3 describes the connection between the serial and parallel portions of the code. Section 4.4 examines the major algorithms and numerical methods employed by the numerical simulator.

4.1 The Physical Model

In order to implement a physical simulator, it is necessary to abstract the relevant physics sufficiently to represent them on a computer and mathematically model them. This section describes the abstractions employed by the simulator for this purpose. Section 4.1.1 examines how viral coat proteins are modeled and discusses what con-

figurable parameters are available to users. Section 4.1.2 describes the mathematical representation of the binding interactions between coat proteins. Section 4.1.3 discusses the representation of the environment in which virus shells assemble.

4.1.1 Coat Proteins

The simulator models the interactions of individual particles, also referred to as nodes, which represent single viral coat proteins. A node has a characteristic size, shape, mass, and bond configuration. Optionally, a node can undergo conformational shifts, allowing it to alter these properties probabilistically. In order to support these characteristics, the simulator uses three different types of nodes: single, complex, and variable. These three node types allow a representation of coat proteins that more accurately reflects their true structure than was possible with the prior simulator.

A single node is the basic unit of node construction. It consists of a sphere with a specified mass, radius, and configuration of edges projecting from its center. Figure 4-1 shows an example of a single node possessing four edges. In defining a single node, a user specifies its mass, radius, the end points of its edges relative to the center of the node, the types of these edges, and "up" vectors for each of the edges, which control how they attach to other edges. The edge parameters are explained in Section 4.1.2. A coat protein can be modeled by a single node, or can be constructed from separate single nodes by using the other two types of nodes.

Complex nodes are built from nodes of other types. A complex node is defined by a set of node types, each having a specified offset and rotation within the complex. These sub-nodes are physically modeled as if they were rigidly connected to each other, causing the entire complex to behave as one solid particle. It is not, however, necessary for the sub-nodes to be in contact with each other, although for physically reasonable particles they generally will be. Figure 4-2 shows an example of a complex node. This node is made up of three single nodes, each of which has a distinct relative position, radius, and bond configuration.

Variable nodes are used to implement conformational shifting. They are defined by a set of other node types, each with a characteristic energy. A variable node shifts

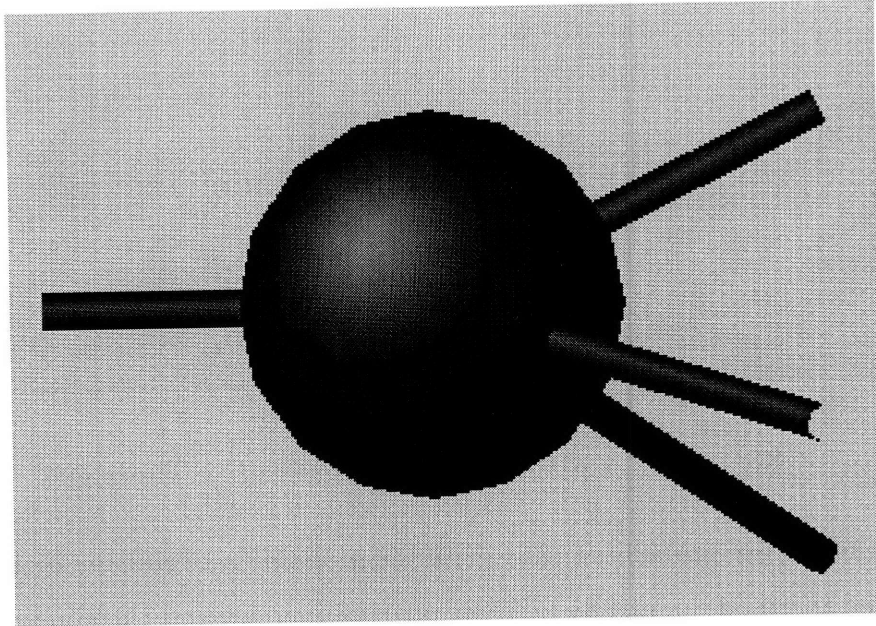


Figure 4-1: A Single Node

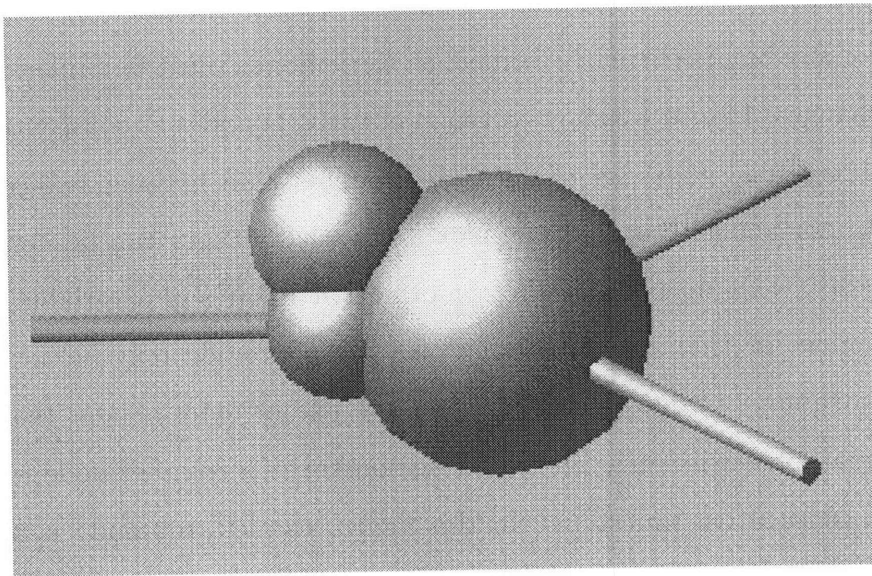


Figure 4-2: A Complex Node

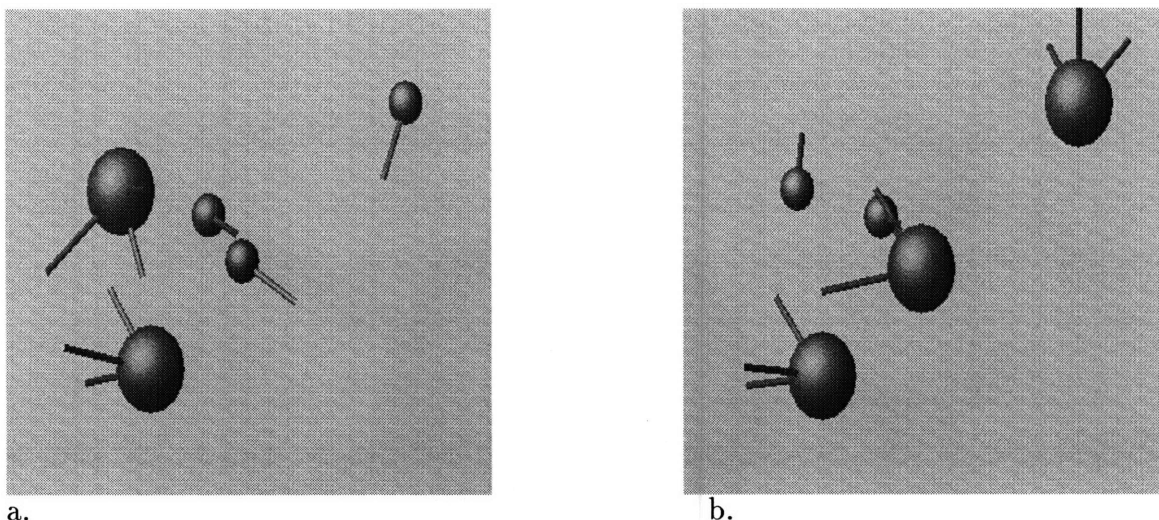

a. b.

Figure 4-3: Variable Nodes at Consecutive Timesteps

between these types probabilistically, with the probability of shifting to different types determined thermodynamically by their relative energies and the simulation temperature. The probability of a particle with m states occupying a particular state i is given by the following formula:

$$p_i = \frac{e^{-\frac{E_i}{RT}}}{\sum_{j=1}^{m} e^{-\frac{E_j}{RT}}}$$

where E_k is the energy of state k, T is the absolute temperature, and R is a constant of proportionality. The energy of the current state is modified by adding to it the energies of any bonds currently formed, as those bonds must be broken if the conformation changes. Figure 4-3 shows a group of variable nodes at consecutive timesteps. In these images, all nodes are of the same type, but shift between two conformations.

One important note is that complex and variable types can be defined in terms of each other. This means that the conformational states of a variable node may be complex nodes or that one subsection of a complex node may be a variable node, allowing a subunit to have a stable region and one or more regions that are capable of conformational switching. Figure 4-4 demonstrates a complex node with a variable region at consecutive timesteps. In this figure, the left sub-node is fixed, but the right shifts between two states. The capacity to define variable and complex nodes in terms of one another also allows one unit of a complex node to be a smaller complex node, or one state of a variable node to be a variable node; while the ability to give

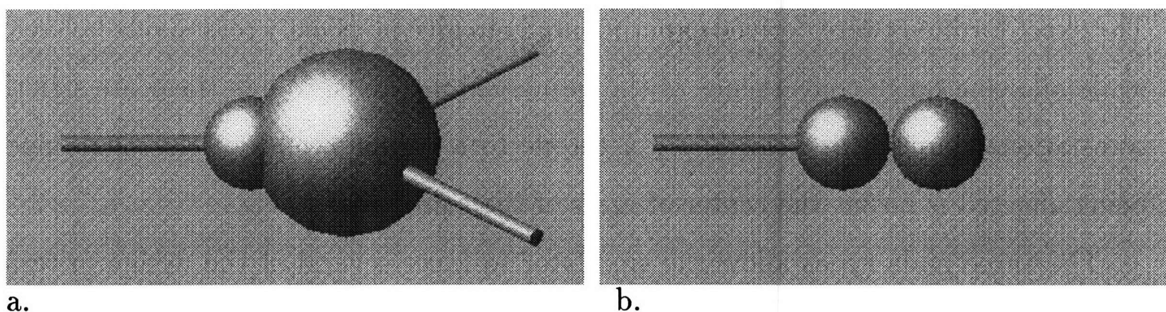


Figure 4-4: A Complex Node Containing a Variable Region

a complex node a complex sub-node or a variable node a variable conformation does not add functionality to the simulator, it can be convenient in defining node types.

In the simulation, all nodes have a full six degrees of freedom. This means that the state of a node, excluding conformation and edge states, consists of twelve real values: three defining translation from the origin, three defining a rotation, three defining velocity, and three defining angular velocity. These values are influenced by two principle types of forces acting on the nodes. The first of these are bond forces, which are described in Section 4.1.2. Second are collision forces, exerted either by collisions with other nodes or by collisions with the boundaries of the simulation region, which are treated as elastic walls. Collisions occur whenever the spheres representing two single nodes overlap, or one sphere is partially outside the edge of the simulation. The direction of such a force is on the vector along which colliding objects have their greatest overlap. For colliding nodes, this is always the vector between the centers of the two colliding single nodes, while for wall collisions it is perpendicular to the plane of the wall. For wall collisions, the magnitude of the force is given by:

$min\{Cm\frac{1-\frac{o}{r}}{(\frac{o}{r})^2}, 12Cm\}$

where C is a constant, currently set at fifty kdalton·nm/μsec^2, m is the mass of the node, o is the overlap between the node and the wall, and r is the radius of the node. For collisions between two nodes with vector $\vec{d}$ between them, the magnitude of the force on each is given by:

$min\{C\frac{(r_1+r_2)^2-(r_1+r_2)||\vec{d}||}{||\vec{d}||^2}, 2C\}$

where C is defined as for wall collisions and r_1 and r_2 are the radii of the two nodes.

The exact formulas were derived experimentally to give physically reasonable behavior while allowing quick convergence of the numerical methods. While these are strictly translational forces, they can apply a torque to a complex node if the single node being affected is not at the center of mass of the complex.

The three node types allow the simulator to meet several of the design criteria. By using six degrees of freedom, the simulator can support the more realistic model of space and time required. Users can control sizes, masses, and bond configurations of nodes through the single node parameters. Complex nodes allow modeling of different node shapes, while variable nodes provide the necessary support for conformational switching. The recursive design offers a model of nodes that is both versatile and easy to use. Finally, representing nodes as unions of spheres allows for efficient processing by simplifying calculations of collisions, as it is only necessary to measure the distance between the centers of two spheres to determine if they are overlapping.

4.1.2 Binding Interactions

Binding interactions are modeled with edges, which can connect nodes by forming bonds to other edges. An edge type is defined by four general properties: its strength, its energies, its tolerances, and the other edges to which it can bind. Furthermore, specific instantiations of an edge type are included as part of a single node description and have two additional properties: the position of the end of the edge relative to the center of the single node, and an "up" vector, whose use is described below.

Whether or not two unbound edges will bind to each other is determined by three tests. First, one must be listed as an acceptable match for the other in the list of edge types to which the other can bind. Second, they must be within the allowed tolerances of each other. Tolerances specify a maximal distance between the ends of the two edges and a maximal angular difference between the directions of the two edges. If both of these tests are passed, then the bond will have a probability of forming determined by the energies of the edges. The probability of a bond forming is given in terms of a reaction energy e_r as $\frac{e^{-\frac{e_r}{RT}}}{1+e^{-\frac{e_r}{RT}}}$, where T is the absolute temperature and R is a constant of proportionality. Once formed, the bond has a probability of breaking

given by a second energy e_d as $\frac{e^{-\frac{e_d}{RT}}}{1+e^{-\frac{e_d}{RT}}}$, with R and T defined as above. Setting $e_r = e_d$ gives a simple approximation to realistic kinetics in which the long term probability of a bond being formed will be what statistical thermodynamics predicts. Using different values for e_r and e_d provides a more complicated but realistic model incorporating the idea of an activation energy.

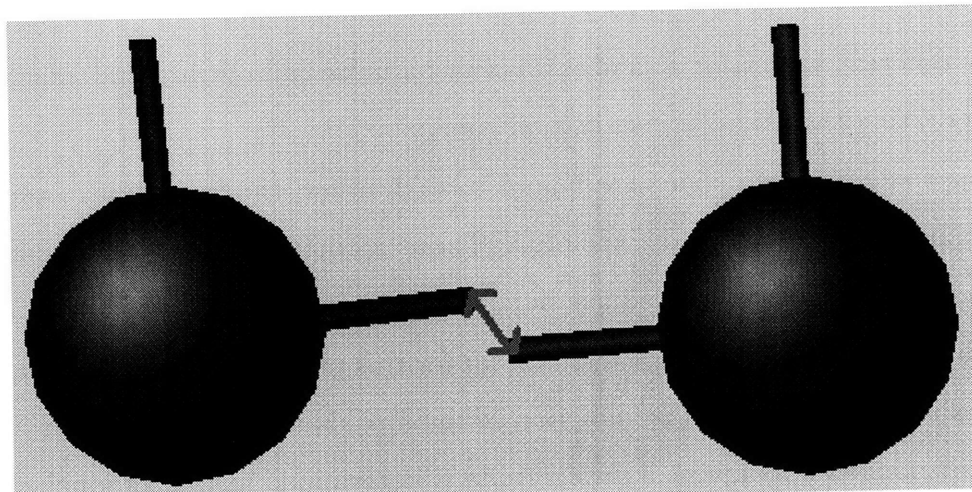
Once edges of two nodes are bound to each other, the forces they exert on each other are modeled by three springs. Those springs' constants, representing the strength of the bond, are defined by the edge type. This model is based on that used by Muir[17], in the original simulator for this project. The first spring produces a translational force, $\vec{T}_t$. The direction of the force is the vector between the ends of the two bonded edges, and its magnitude is determined by the product of the first spring constant, k_t, and the distance between the ends of the edges. For nodes n_1 and n_2 with edge ends $\vec{e}_1$ and $\vec{e}_2$, this gives the following equation for the force on n_1: $\vec{T}_t = k_t(\vec{e}_2 - \vec{e}_1)$.

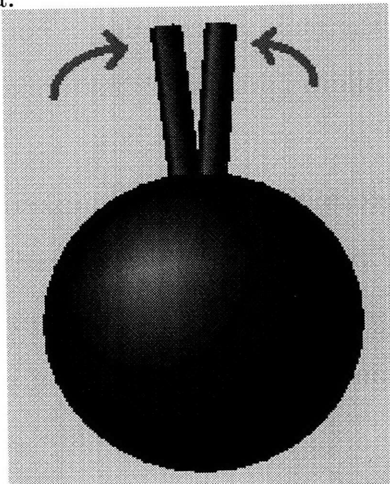
The second spring constant, k_r, is used to determine a torque, $\vec{O}_r$, exerted around a bond, and ensures that nodes are rotationally positioned correctly relative to each other. Two bonds are ideally rotationally positioned when their up vectors are parallel. For up vectors $\hat{u}_1$ and $\hat{u}_2$ and edge direction $\hat{d}_1$, this torque is given by: $\vec{O}_r = k_r[(\hat{u}_1 \times \hat{u}_2) \cdot \hat{d}_1]\hat{d}_1$.

The final spring constant, k_s, determines a "straightening" torque, $\vec{O}_s$, that forces two bonded edges towards being anti-parallel to each other. For edge directions $\hat{d}_1$ and $\hat{d}_2$, this torque is given by: $\vec{O}_s = k_s\sqrt{0.5 + 0.5(\hat{d}_1 \cdot \hat{d}_2)}\frac{\hat{d}_2 \times \hat{d}_1}{||\hat{d}_2 \times \hat{d}_2||}$.

These three forces are illustrated in Figure 4-5. This figure shows the same pair of nodes from three views. Figure 4-5a shows how $\vec{T}_t$ forces the endpoints of the edges together. Figure 4-5b shows a view of the nodes along the line connecting their centers. It demonstrates how $\vec{O}_r$ rotates the nodes around their bonded edges so their up vectors will be parallel. Figure 4-5c shows the direction of $\vec{O}_s$ on each node in straightening the bond.

This model of binding helps to meet the design goals in several ways. It pro-


a.


b.

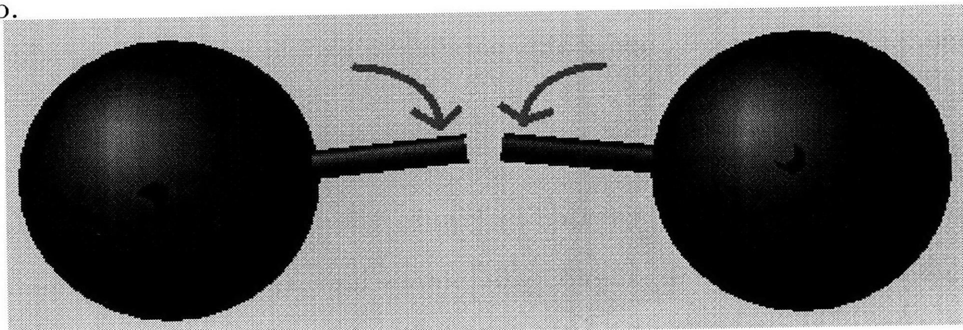

c.

Figure 4-5: Bond Forces Acting Between Two Nodes

vides a realistic model of the kinetics of binding. It further gives users considerable latitude for varying relevant parameters. It is also easy to understand and use in practice. Finally, the computational cost of processing it is small enough to allow high performance.

4.1.3 The Environment

The environment in which particles move around provides a model for the solution in which virus shells assemble. This environment has several user-definable properties. Two of these describe physical properties of the solution: temperature, which influences thermodynamic probabilities and the kinetic energies of the particles, and damping, which represents the viscosity of the solution. The effect of temperature on conformational switching is described in Section 4.1.1, on bonding in Section 4.1.2, and on kinetic energies in Section 4.3.3. Damping applies a force to each particle opposite the direction of its velocity and a torque opposite the direction of its rotation. The exact magnitude of these forces should ideally depend on the surface area of the particle in the direction of its velocity. However, this would be expensive to calculate, so it is approximated as the two-thirds power of mass, giving the following formulas for damping force, $\vec{F_d}$, and damping torque, $\vec{T_d}$:

$\vec{F_d} = -dM^{2/3}\vec{v}$

$\vec{T_d} = -dO^{2/3}\vec{a}$

where M is the mass of the node, O is its moment of inertia, $\vec{v}$ is its velocity, $\vec{a}$ is angular velocity, and d is the damping constant.

In addition to these physical properties are three other configurable parameters. The first of these, the timestep, defines the basic time division needed for a discrete approximation to the problem. Bonds and conformations are updated probabilistically at the beginning of each timestep, and the timestep serves as the base subdivision for the numerical methods, described in Section 4.3.1. The second parameter, tolerance, provides a measure of desired accuracy. The numerical methods use an adaptive step-size based on the user-specified timestep to achieve the specified accuracy on each step. The final parameter, ε, is used in controlling small round-off errors and

represents a value considered to be approximately zero. In must be user-specified based on the magnitudes of the other parameters.

This model of the environment helps to meet some design requirements. Having several parameters gives the user a means of configuring the simulator for a variety of different problems. However, there are few enough parameters that it is easy to use. Furthermore, user control of the timestep and tolerance allows for faster processing, as the simulator need not do more work than the user requires.

4.2 The User Interface

The user interface is the set of systems through which the user sends commands to the numerical simulator and receives back information from it. This section describes the major systems involved and explains how they meet the design goals. Section 4.2.1 covers the command interpreter, which provides the underlying control for the user interface. Section 4.2.2 describes the graphical interface. Section 4.2.3 discusses the graphics display. Section 4.2.4 examines the controller, a set of routines controlling many aspects of the simulator's behavior. Section 4.2.5 discusses an alternate controller. Section 4.2.6 describes the different kinds of data files used by the program.

4.2.1 The Command Interpreter

The underlying mechanism for all user interactions with the numerical simulator is a command interpreter running a Lisp-like simulator control language. This language supports many standard Lisp abstractions, including variables, lists and arrays, and procedure definitions. It also provides links to the graphics, numerical aspects of the simulator, and file I/O, as well as primitive support for many common operations. Figure 4-6 provides an example of code written in the control language, illustrating the general syntax. This sample code defines a procedure fib, for calculating Fibonacci numbers, and a procedure fib_array, which takes a parameter n and returns an array filled with the first n Fibonacci numbers. It also defines an auxiliary procedure used by fib_array. This example shows only a few of the features of the control language.

```
(define fib (procedure (n)
  (if (<= n 1)
       1
       (+ (fib (- n 1)) (fib (- n 2))))))

(define fibarray_aux (procedure (tmp n max)
  (if (>= n max)
       tmp
       (set_array (fibarray_aux tmp (+ n 1) max) n (fib n)))))

(define fibarray (procedure (max)
  (if (<= max 0)
       (make_error "Index out of bounds in fibarray")
       (fibarray_aux (make_array max) 0 max))))
```

Figure 4-6: A Sample of Code in the Simulator Control Language

Detailed specifications of the language are included in Appendix A.

Providing a fully programmable control language for the simulator allows users considerable latitude for controlling the behavior of a simulation where that latitude is required. In addition, by writing many of the control structures in the control language, it is possible to allow significant reconfiguration of the program at run-time by allowing users to load replacement control structures. This is an important consideration for a program which is meant to be distributed to people who are not experienced programmers and must accomplish a wide variety of tasks, some not even considered at the time the code was developed. The versatility of this becomes more apparent in the discussions of the alternate controller in Section 4.2.5 and data files in Section 4.2.6. Furthermore, the use of a Lisp-like language allows for efficient parsing and promotes ease of use through a simple syntax.

4.2.2 The Graphical Interface

Normal user interactions are not directly through the control language, but through a graphical user interface that serves as a front end to the command interpreter. The main window of the graphical user interface is shown in Figure 4-7. This interface

Figure 4-7: The Graphical User Interface

consists of a feedback box and fourteen buttons used for controlling simulations. This section explains the functions of the different components of the graphical interface.

The feedback box, which displays the words "Stepped forward" in the figure, is used to display the results of several of the commands. It can indicate when errors have been encountered, for instance because a rule definition was badly formatted. It also provides the outputs of user queries on nodes. Finally, it can signal that a command has finished executing, as "Stepped forward" does in Figure 4-7.

The four buttons beneath the feedback box control the graphics display. The "zoom in" and "zoom out" buttons increase and decrease the magnification of the graphics display. The "rotate" button allows the user to rotate the display by a specified angle around one of the three coordinate axes. Finally, the "recenter" button is used to reposition the focal point of the graphics display by having the user specify its new x, y, and z coordinates.

The "add" and "delete" buttons control the number of nodes in a simulation. The "add" button allows the user two options. The first is to add a new node with a specific type, position, orientation, velocity, and rotational velocity. This is useful when attempting to control fine details of a simulation, for example by testing the effect of bombarding one particle with another of known velocity and orientation. The second option is to add a specified number of nodes to the simulation with random positions and velocities. The latter is more useful when using a large number of nodes to model a solution of particles. The "delete" button is used to remove particles from the simulation and also has two options. With the first option, delete is given the index of an individual node and deletes that particular node. This is generally useful

in simulations of a small number of particles. The other option, useful for larger simulations, deletes all particles within a rectangular region in the workspace.

The "load" and "save" buttons control access to files. The "save" button writes out the contents of the current simulation, including workspace parameters, defined node and edge types, the states of all nodes present, what conformations they are in, and how they are bound to each other. It prompts the user for the name of a file into which this data is saved. The "load" button reads in a data file by prompting the user for the name of the file then passing it to the interpreter. The types of files that can be loaded and how they are processed is described in more detail in the Section 4.2.6.

The "set" button tests and sets the values of several user-definable simulation parameters. Some of these, i.e. temperature, damping, timestep, tolerance, and ε, were described Section 4.1.3. Of the others, the first, steps_per_update, controls the number of timesteps the numerical methods take each time they are prompted by the user. This allows the user to limit the number of times the graphics are updated when they are not immediately needed. Another, updates_per_run, instructs the simulator to run for a specific number of steps without further user interaction. Finally, there are two parameters, host and port, that specify how the controller should locate a server machine, which can optionally be used to perform the numerical computations. The meaning of this is further explained in Section 4.3.

The "query" button asks the simulator for information on the positions and velocities of nodes. It functions similarly to the "delete" button, allowing a user to specify either a particular particle or a rectangular region. If a single particle is specified, then data for that particle is printed in the feedback box. In the case of a region, average values for that region are printed.

The "direct" button provides a direct link to the command interpreter. This button produces a window that allows the user to type commands directly to the interpreter and prints their results. This is useful when the user requires greater functionality than the graphical interface provides, but does not want to create and load a data file.

The "step" button advances the simulator in time. In normal operations, pressing the "step" button causes three actions to be undertaken. First, the numerical model advances in time according to the parameters the user has set. Next, the command interpreter updates its local data set based on the results of advancing the time. Finally, the interpreter updates the graphics display in accordance with the new data. These three actions may be repeated one or more times, depending on the simulation parameters.

The "restart" button reinitializes the simulator data and establishes a new connection to a server. It erases all existing nodes and all node and edge types. It also sets many simulation parameters back to their default values. Finally, it establishes a new connection to a server based on the values of the host and port variables. Host specifies a particular machine to which the interface should connect, while port is the value of a port on that machine at which the interface looks for a server.

The "quit" button exits the user interface. It breaks off the connection to the server, if any, and terminates the program. It shuts down only the user interface, however, not the server machine running the numerical simulator.

One important note is that, since the graphical interface is a front end to the control language, the effects of its commands can, for the most part, be reprogrammed at run time. Although reprogramming commands can cause buttons to behave completely differently than their defaults, this feature is meant to allow commands to be slightly modified without affecting their basic purpose. The utility of this is described in more detail in Sections 4.2.4 and 4.2.5.

The use of a graphical user interface as a front end to the command interpreter is meant primarily to make the program easier to use. The graphical interface allows faster access and requires much less knowledge of the program to use than a purely text-based interface would. However, a simple interface must necessarily reduce versatility. For that reason, the user is provided a means of accessing the underlying command interpreter; this gives the user a high degree of control when necessary, while still allowing easy access for those users who do not require such control. Furthermore, providing a graphical interface that can be reprogrammed supports the

goal of configurability.

4.2.3 The Graphics Display

The graphics display is the major means by which the user can observe the progress of a simulation. It provides an easily understood visualization of the simulated nodes and their relative positions and orientations. Figure 4-8 shows a sample graphics display of a simulation containing a small number of nodes. Each single node is drawn as a sphere, with edges represented as cylinders radiating from the centers of the spheres. Nodes are colored according to their type, with the color, in the case of complex and variable nodes, set by the type of the complex or variable, rather than the sub-types that make it up. Therefore, all conformations of a given variable node have the same color, as do all sub-nodes of a given complex node. The box drawn around the display shows the position of the walls around the simulation workspace. Since the graphics are run through the interpreter, they can be modified to display the workspace differently, although this flexibility is not currently used.

The graphics shown here are produced through the OpenGL[20] rendering package, but this is not the only option. OpenGL provides a versatile model of graphics, supporting shading, z-buffering, and a variety of surface and lighting models; all of this makes the graphical output much easier to understand. However, of the available machines, only Silicon Graphics Indigos support OpenGL. Therefore, the graphics can alternately be viewed with Vogle[10], a public-domain package that draws three-dimensional surfaces as unshaded polygons. This provides a lower quality of output, but increases portability of the code.

The graphics system helps the simulator to meet several design requirements. Graphical output is easier to understand than numerical output, making the simulator easy to use. Furthermore, running graphics through the interpreter allows it to be configurable. The use of two options for rendering, OpenGL or Vogle, allows the code to take advantage of available hardware while still having high quality output on hardware that supports it. Finally, the graphical representation is sufficiently simple that it should not be a bottleneck.

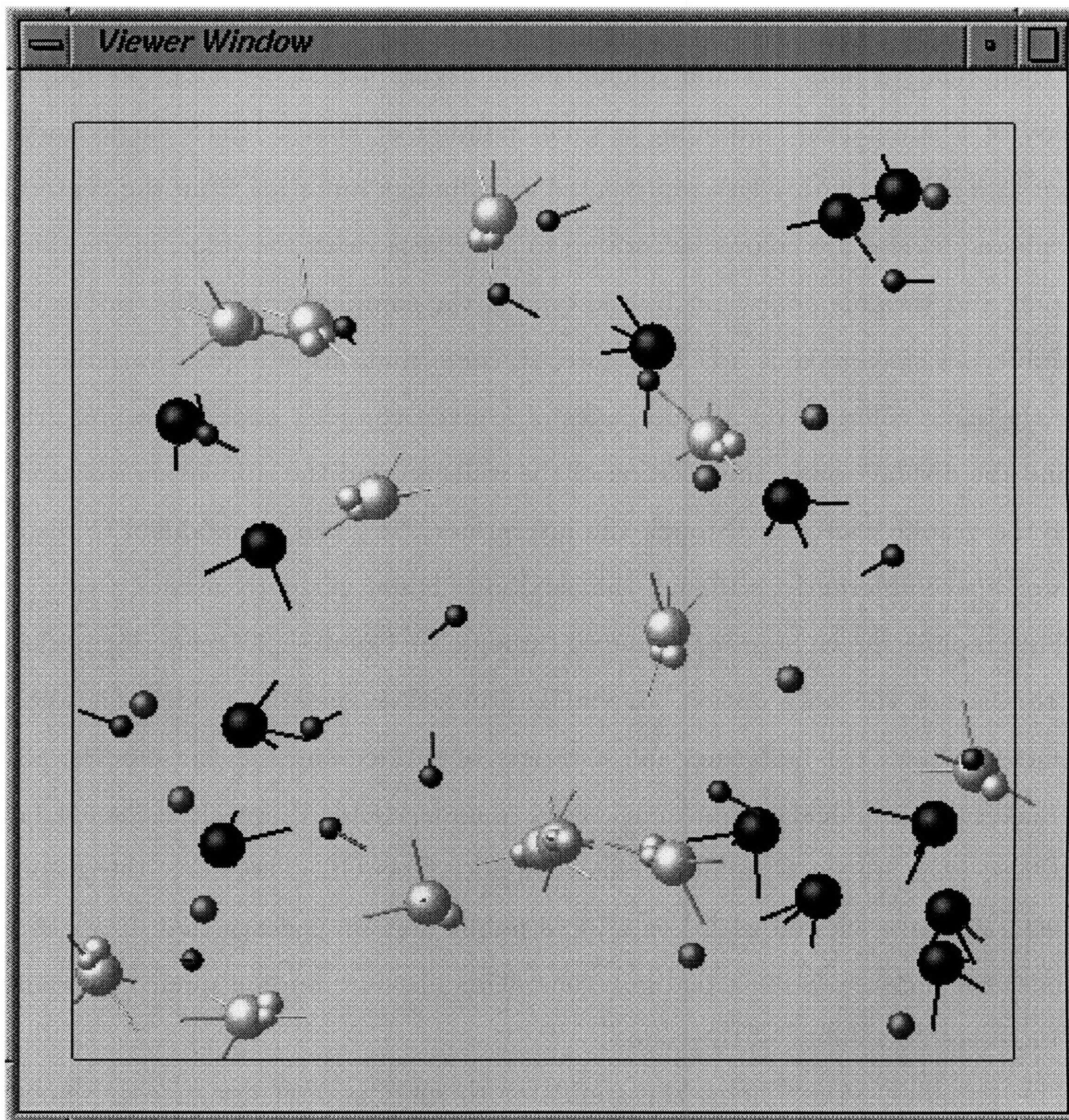


Figure 4-8: Graphics Display of a Sample Workspace

4.2.4 The Controller

The controller determines how to respond to commands issued through the graphical interface and maintains much of the data on the state of the simulation. The controller itself is written in the simulator control language and run through the command interpreter. The controller directs the graphical interface through a series of "hooks," procedures activated by use of certain buttons or text commands or other special conditions. This section describes the major functions of the controller and explains how it helps to meet the design goals. Descriptions of all hooks and specifications of their default behaviors are provided in the appendix, in Section A.2.

The most complicated features of the controller concern adding and deleting nodes and edges. The controller maintains copies of all node and edge types defined on the numerical simulator. It provides routines that define new types and update the numerical simulator on their status. It also provides similar routines for adding new nodes and attaching edges which update the local data and communicate the changes to the numerical simulator.

The controller is also responsible for updating the graphics display. In order to draw the display, the controller first queries the numerical simulator to update the local data. It then clears the graphics buffer and draws all nodes present by placing one sphere per single node and one cylinder per edge. Finally, it draws the bounding box and displays the updated buffer.

The controller additionally determines how to advance time. The model implemented is very simple. The controller instructs the numerical simulator to advance the simulation time by a prespecified amount. It then calls the routines described in the previous paragraph for updating the display. This is repeated a predefined number of times. This allows the simulator to run for a time without additional user input.

The controller also processes some user queries. The controller receives data on individual nodes from the numerical simulator and formats this data into a user-readable form for display in the user interface window. Additionally, when average data on a region of the workspace is required, the controller determines which nodes

are in the specified region and averages their data.

Finally, the controller is responsible for saving the state of the simulation to a file when requested. The simulator uses a series of queries to the numerical simulator to determine the values of various simulation parameters and writes them into the save file. It then adds definitions for all node and edge types. Finally, it uses additional queries to establish the states of the nodes and add them to the save file.

Having a controller written in the simulator control language supports the goal of versatility. Small modifications to the controller can be made easily and stored in separate data files. These modifications can then be loaded at run time, allowing easy reconfiguration of the simulator. Larger modifications are more difficult to write; however, because of the generality of the control language, they can considerably extend the capabilities of the controller.

4.2.5 The Alternate Controller

The alternate controller is a partial replacement of the code used by the controller described in Section 4.2.4. It is useful both in its own right and as an illustration of the power of the simulator. The alternate controller implements a model of assembly much like that used by Muir[17], but retaining the more realistic models of reaction kinetics and time and more versatile model of nodes employed by the new simulator. In order to accomplish this, it redefines a single hook used by the simulator: the do_step_forward hook activated by pressing the "step" button. This section briefly describes shell assembly under the alternate controller.

The alternate controller adds nodes one at a time to the simulation. If no nodes are present, the simulator begins by adding one node chosen from a list of types specified by the user. If a node is already present, it instead probabilistically selects one unbound edge from the available nodes in accordance with the relative energies of the unbound edges. It then chooses a node type to attach to that edge, inserts it into the simulation with the correct position and orientation, and forms the bond. Finally, the alternate controller advances time by a prespecified amount and updates the display. This is repeated a predefined number of times before the alternate controller returns

control to the user.

The alternate controller is useful for refining node properties before using the refined nodes in a full simulation. It allows easy detection of mistakes in rules, which might be difficult to see in a simulation of free-floating particles. It also permits testing of the stability of a completed shell without having to consider the effects of stability on the growth process. This allows examination of problems that might make a model infeasible but cannot easily be tested in a general simulation.

The alternate controller helps to fulfill the design goals in several ways. It simplifies the process of designing nodes. It also speeds up this process by providing a quick test to screen out many unusable models without running a full simulation. It further supports the versatility of the simulator by allowing testing of some kinds of problems that could not be explored with the default controller. The alternate controller also demonstrates the configurability of the simulator; it shows that a user can, at run time, significantly alter the behavior of the simulator by loading one file.

4.2.6 Data Files

Data files are used for three basic purposes in the simulator: as rules files, save files, and control files. Rules files define a set of node and edge types, set up some simulation parameters, and often create a set of starting nodes; typically, this is used to establish the initial state for a simulation. Save files store the state of a simulation in progress, as described in Section 4.2.2. Control files set up control structures within the command interpreter that might be useful in running a particular simulation; an example of a control file is the alternate controller of Section 4.2.5. This section describes the different types of data files and discusses some details of their implementation.

The term rules file can refer to any file loaded preparatory to running a simulation for setting up the parameters of that simulation. A simple rules file is shown in Figure 4-9. The first five lines of this file set several of the simulation parameters. The remainder define some node and edge types. For example, line six creates an edge type which connects to its own type, has all spring constants ten, both binding

```
(resize_workspace 100 100 100)
(set_constant "damping" .5)
(set_constant "timestep" 0.25)
(set_constant "tolerance" .5)
(set_constant "temperature" 300)
(add_edge_type (make_list 0) 10 10 10 -5000 -5000 25.0 55)
(add_single_node_type 10 0 3 (quote ()) (quote ()) (quote ()))
(add_single_node_type 5 0 15
  (make_list (make_list 0 10 20) (make_list 10 20 30))
  (make_list (make_list 1 0 -1) (make_list 1 -1 0))
  (make_list 0 0))
(add_complex_node_type
  (make_list (make_list 0 0 0) (make_list 0 2 0))
  (make_list (make_list 1 1 1) (make_list 2 2 2))
  (make_list 0 0))
(add_variable_node_type (make_list 0.0 0.0) (make_list 1 2))
```

Figure 4-9: A Sample Rules File

energies -5000, a distance tolerance of twenty-five, and and angle tolerance of fifty-five. The file then defines two single node types and a complex and a variable node type in terms of them.

Save files store the state of a simulation at the time it is saved. They are usually produced by pressing the "save" button. They can be reloaded later to continue the simulation from that point. A save file therefore must contain all information needed to define the state of a simulation. A sample save file is shown in Figure 4-10. The file first sets the simulation parameters and defines data types, as in a rules file. It next specifies the types, positions, velocities, rotations, and angular velocities of the nodes in the simulation. After that, it provides the states of the conformations of all variable nodes. Finally, the file specifies the pairs of bonded edges.

Control files, other than the controller, are not needed for normal operations but can be useful in tailoring a simulation to a specific task. They supplement or replace the control structures provided by the controller. A sample control file is shown in Figure 4-11. This file redefines draw_workspace, one of the hooks to the graphics routines, and provides two auxiliary procedures it requires. The sample file redefines

```
(resize_workspace 100 100 100)
(set updates_per_run 1)
(set_constant "e0" 1)
(set_constant "u0" 1)
(set_constant "g" 1)
(set_constant "timestep" 0.25)
(set_constant "tolerance" 0.5)
(set_constant "temperature" 300)
(set_constant "steps_per_update" 1)
(set_constant "epsilon" 1e-08)
(set_constant "damping" 0.5)
(add_edge_type (make_list 0 ) 10 10 10 -5000 -5000 25 55)
(add_single_node_type 10 0 3 (make_list ) (make_list ) (make_list ))
(add_single_node_type 5 0 15 (make_list (make_list 0 10 20) (make_list 10 20 30) )
(make_list (make_list 1 0 -1) (make_list 1 -1 0) ) (make_list 0 0 ))
(add_complex_node_type (make_list (make_list 0 0 0) (make_list 0 2 0) ) (make_list
(make_list 1 1 1) (make_list 2 2 2) ) (make_list 0 0 ))
(add_variable_node_type (make_list 0 0 ) (make_list 1 2 ))
(add_node 2 (make_list 14.848 32.2073 46.1229) (make_list 15.1462 -0.485406 19.9144)
(make_list -0.0419502 0.751216 -1.81328) (make_list -4.14381 15.6335 -14.6084))
(add_node 3 (make_list -43.086 -41.5043 12.2966) (make_list 30.9144 -3.24541 21.6224)
(make_list -1.04092 0.320605 0.38598) (make_list 1.11564 -2.00581 2.2245))
(add_node 3 (make_list -20.8355 -35.3789 -41.3804) (make_list -1.51766 -2.26039
9.94266) (make_list -0.624382 -0.824785 0.188249) (make_list -9.12478 -9.01502
5.19569))
(set_conformation 0 0)
(set_conformation 1 0)
(add_edge 0 2)
(add_edge 1 3)
```

Figure 4-10: A Sample Save File

```
(define draw_single_node_new (procedure (n x y z) (sequence
  (define node (get_array the_singles n))
  (define type (get_array the_node_types (single_type node)))
  (make_sphere x y z (single_node_type_r type)
       (first (get_array the_nodes (single_parent node)))))))

(define draw_workspace_aux_new (procedure (data n) (sequence
  (if (>= n 0)
       (sequence
         (draw_single_node_new
              (get_array (first data) n)
                (first (get_array (first (rest data)) n))
                (first (rest (get_array (first (rest data)) n)))
                (first (rest (rest (get_array (first (rest data)) n)))))
         (draw_workspace_aux_new data (- n 1)))))))

(define draw_workspace (procedure () (sequence
  (errortrap
       (draw_workspace_aux_new
         (rest workspace_data)
         (- (first workspace_data) 1)))
  (draw_box)
  (redraw 7))))
```

Figure 4-11: A Sample Control File

the procedure for drawing the workspace so edges would not be drawn.

It is important to note that the distinction in file types is based only on how and where they are used, not on what they contain. In fact, all data files are made up of sequences of commands in the control language. Loading any of them consists of running them through the command interpreter as if they had been typed in by a user.

This method of implementing data files supports the design goals by promoting configurability of the code. The use of a general programming language for rules files allows users to define complex rule constructs and automate some aspects of rule specification. Furthermore, because of the generality of the control language, save files can be easily customized for particular applications. The format of save files also has the advantage of being easily understood and edited by a user. Finally, supporting

control files allows a simple model for storing and reloading customizations that are likely to be used more than once.

4.3 The Serial/Parallel Interface

In order to take advantage of the available parallel machine, it was necessary to develop a way to allow user interactions with the machine. It was decided to develop the code in two pieces: a user interface running on a workstation and a numerical processing section running on a server machine. Typically, this server is the CM-5; however, it could be another machine, including the same machine that is running the user interface. These two parts of the simulator communicate over a network connection by means of a series of commands and queries issued by the user interface and answered by the server.

In order to run a simulation, the interface must establish a connection to a server machine. The user first specifies a desired machine, which must be running the numerical simulator. The interface then connects to that machine. User commands are interpreted by the controller, which issues it own commands and queries to the server machine as necessary. These either instruct the server to update its internal data or ask about the status of that data. The controller uses the server's responses to these queries to update its own local data which is in turn output as a graphics display or in a user-readable text form.

This particular model is meant to improve performance and make efficient use of resources. Running the user interface on a local machine has the advantage of avoiding sending large amounts of graphical data over the network, since rendering is done locally. It further allows many serial parts of the program to run on a local machine. These routines are unlikely to benefit from being run on the CM-5 and could greatly increase memory usage on that machine, where memory is much more constrained.

4.4 Major Algorithms and Numerical Methods

This section describes the techniques used to perform the most computationally intensive aspects of the simulation. Section 4.4.1 discusses the numerical methods used to approximate the evolution of a simulation over time. Section 4.4.2 describes the parallel routines used to calculate the forces on the particles and advance them by one timestep. Section 4.4.3 examines a technique used by the program to overcome difficulties in maintaining realistic thermodynamics over time.

4.4.1 Approximation of the ODE

Perhaps the most crucial operation of the simulator is advancing the particles forward in time. In understanding how to approach this problem, it is helpful to view it as evaluating a system of differential equations. Specifically, for each degree of freedom there are two corresponding differential equations: the first defining velocity as the derivative of displacement, and the second defining acceleration as the derivative of velocity. Then, calculating the state of the simulation at a future time requires determining the forces on the particles in order to find the accelerations, and solving the resulting differential equations. Given this understanding, it is only necessary to choose an effective numerical method for solving the differential equations. The particular approach chosen for this project is the Euler method with Richardson extrapolation.

The basic Euler method is the simplest technique for computing a numerical solution to an ordinary differential equation. In the one variable case, a differential equation $y(t) = f(y, t)$ can be approximated at a time $t + \Delta t$ given the value at time t, using the following equation:

$y_{n+1} = y_n + \Delta t f(y_n, t)$

where $y_n = y(t)$ and y_{n+1} is an approximation to $y(t + \Delta t)$. This process can be repeated starting from a single value to find approximate solutions at many subsequent times. The Euler method has first order accuracy in time, meaning its error after a particular amount of time grows linearly with Δt.

Richardson extrapolation is a technique for improving the accuracy of other approximation methods. The technique uses approximate values generated at different step sizes to generate a higher order approximation. It requires that the error, e, generated by the approximation technique be a function of the step size, Δt, of the form:

$$e(\Delta t) = k_1 \Delta t + k_2 \Delta t^2 + k_3 \Delta t^3 + \dots$$

where each k_i is a constant. This criterion is approximately true for the Euler method, provided the function being modeled and its derivatives are continuous. Richardson extrapolation uses a linear combination of two approximations with different timesteps to eliminate the low order error term of the approximation. Two improved values can then be combined to eliminate the next lower error term, and so on, allowing the approximation to improve exponentially in the number of different step sizes used. The simulator uses extrapolation to refine the approximation to the positions and velocities of the nodes.

The Euler method and Richardson extrapolation are combined with an adaptive timestep method. Using this method, the user-specified timestep provides a base which is successively halved. Each time it generates an improved approximation, the simulator calculates new extrapolated values using the extrapolated values from the previous round. The values for the current round are compared to those of the previous round and if corresponding values from two consecutive rounds are within a specified tolerance for all parameters of a given node, then the timestep is fixed for that node. The result is that the simulator can spend more time on nodes requiring a lot of optimization than on those requiring very little. Once all nodes have reached the specified level of accuracy, the simulator stores the final values and is ready for the next step.

Although this particular method is non-standard, it should be a good choice for this application. Large systems of differential equations are typically solved through the fourth order Runge-Kutta method. However, the Runge-Kutta method requires calculating the forces on each node four times per timestep, versus only once for the Euler method. For this application, calculating forces is computationally very expen-

sive. Therefore, unless the Euler method requires many more steps than the Runge-Kutta method, the Runge-Kutta method would be unlikely to be more work-efficient. For cases requiring only low accuracy, Euler should therefore outperform Runge-Kutta. Further, when high accuracy is needed, Richardson extrapolation quickly allows the chosen technique to surpass the fourth order Runge-Kutta method in order of accuracy. There is a potential pitfall, however. Richardson extrapolation fails to converge when the derivatives of the function being extrapolated are not bounded. Therefore, the extrapolation may not help in modeling collisions, which create discontinuities. Thus, it is difficult to predict how much extrapolation will improve performance in practice.

4.4.2 Advancing Time

In order to efficiently apply the numerical methods just discussed, it is necessary to have a fast technique for performing the function evaluations they require. In the case of a particle simulation, this means computing the forces acting on the particles, updating their velocities based on these forces, and updating their positions based on their velocities. This is accomplished through the use of a parallel, thread-based algorithm for dividing the problem into smaller subproblems that can be handled independently. This section explains why this technique was adopted and describes it in detail.

A thread system allows a computation to be represented as a set of small "threads" of computation whose orders of operation may be only partially specified. Because the final result of the computation does not depend on the threads being completely ordered, it is often possible to run them in parallel. The thread-based code for this simulator was written using Cilk[4], an extension to the C language developed at MIT's Lab for Computer Science. Cilk employs a work-stealing method for allocating threads among processors; that is, it allows idle processors in a multi-processor system to steal threads from busy processors. If a problem is well-partitioned into threads, this can distribute work fairly evenly over all processors in the system. Exploiting work-stealing therefore requires an effective method of domain decomposition, i.e. the

splitting of a problem into smaller subproblems.

The simulator's domain decomposition separates particles according to spatial locality. The algorithm begins by looking at the locations of all particles along one of the coordinate axes and determining the midpoint between the two most distant particles along this axis. It then separates the particles into two groups around this midpoint and creates two separate threads, one responsible for each group of particles. These two threads then recurse on their own groups of particles, splitting along another coordinate axis. The process repeats, cycling through the axes, until the number of particles in the thread under consideration drops below a small constant. At that point, that thread of execution is ready to advance those particles. A sample two-dimensional domain decomposition using this technique for a small set of particles is shown in Figure 4-12. This decomposition divides particles first along the x-axis, then along the y-axis, repeatedly, until they have been separated into boxes of one particle each. In the actual simulator, subdivision currently stops at boxes of five or less, but one is used here to make the decomposition easier to understand.

A spatial decomposition is well-suited to this particular problem, as the probability of two particles exerting forces on each other is closely related to the distance between them. This is because collisions require particles to be in contact and bonds usually only exist between particles that are close together. Therefore, splitting up problems by position reduces the amount of information a thread needs about particles for which it is not responsible. The particular decomposition chosen also has some advantages. Another method by which particles could be split spatially is through the use of oct-trees, which recursively divide the space into identical octants. However, this method has a bad worst case when particles are clustered, in which considerable work can be spent without splitting up the particles at all. This is illustrated in the two-dimensional case in Figure 4-13a. It is expected that particles will cluster in a virus simulation, so this presents a problem. Another approach which would avoid this worst case is to split particles at the median, as with a k-d tree, guaranteeing that the particles will be split into equal or near-equal groups at each step. However, this presents a different problem when particles cluster, as it can group some particles

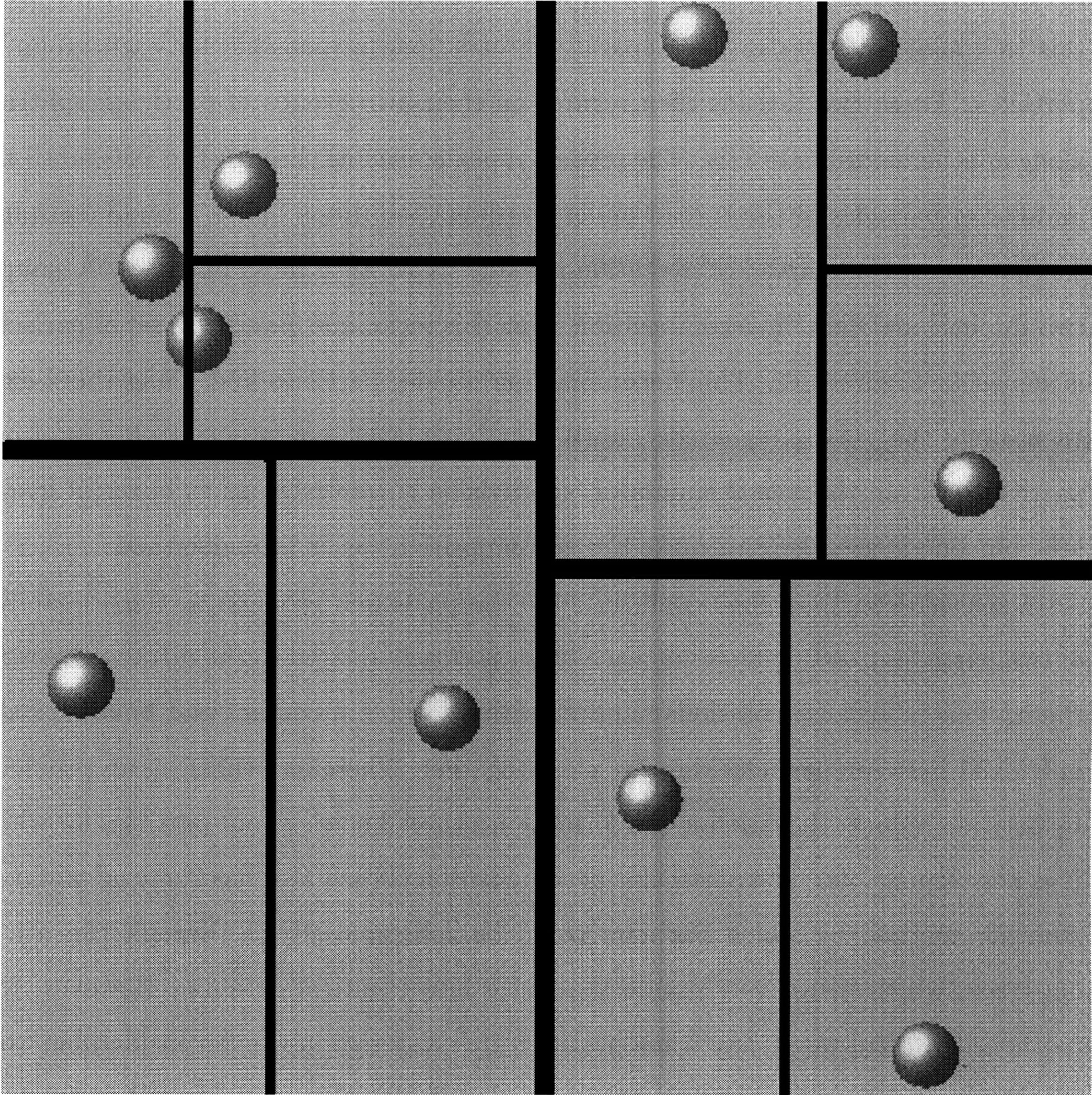


Figure 4-12: A Two-Dimensional Domain Decomposition

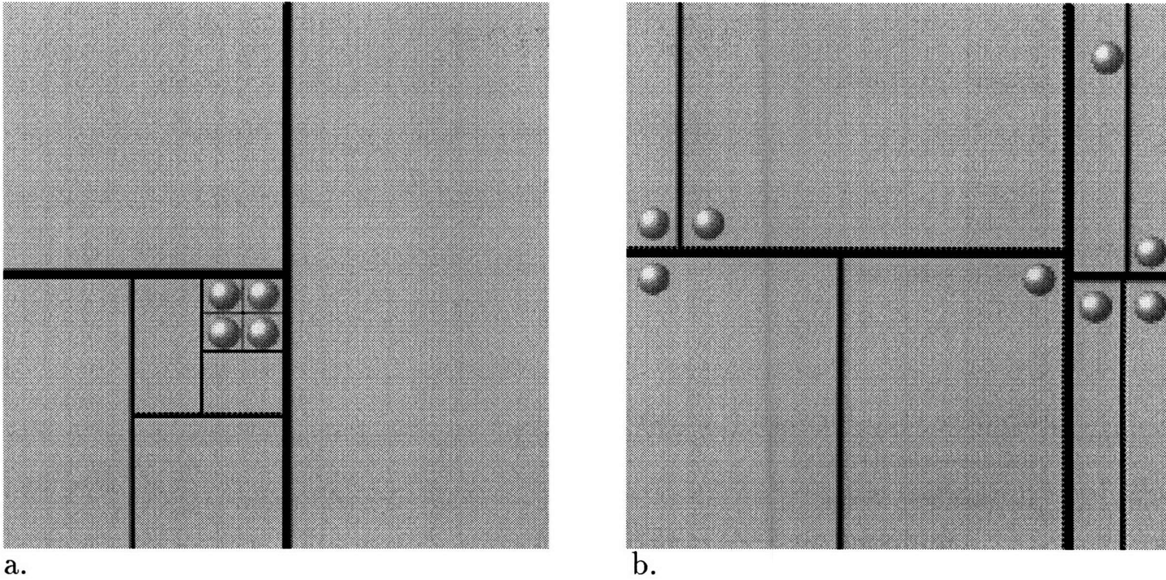


Figure 4-13: Problems Resulting in Poor Decompositions for Quad Trees and K-D Trees

with others spatially very distant from them. This is illustrated in Figure 4-13b. The particular approach chosen can also perform poorly, if the density of particles grows geometrically along one of the axes. However, this seems less likely to occur in practice than the other worst cases; the chosen method also guarantees that the space will have to be split at most as many times as there are particles, even if its worst case occurs.

This decomposition gives rise to a threaded procedure for advancing the simulation time. Pseudo-code for this procedure is shown in Figure 4-14. If the number of nodes in a thread is greater than a constant, k, then the code divides the nodes into left and right subsets as described above. It then spawns two new threads, one for each subset, allowing the subsets to be handled in parallel. When a processor receives a thread containing a small enough number of nodes, it then performs the actual computations for advancing time. First, it computes the forces on the nodes according to the physical model described in Section 4.1. Second, it updates the velocities in accordance with these forces. Third, it updates the positions according to the velocities. Finally, the updated data from the thread must be recombined with that of all other threads.

```
procedure step-forward (nodes)
  axis ← next(axis)
  if (|nodes| > k)
        midpoint ← (max(axis)+min(axis))/2
        left ← ∅
        right ← ∅
        for i ← 1 to |nodes|
          if (node[i].axis < midpoint)
                left ← left ⋃ node[i]
          else
                right ← right ⋃ node[i]
          end
        end
        spawn step-forward (left)
        spawn step-forward (right)
  else
        for i ← 1 to |nodes|
          forces[i] ← find_forces (nodes[i])
          velocity[i] ← find_velocity (velocity[i],forces[i])
          position[i] ← find_position (position[i],velocity[i])
        end
        update_data (position,velocity)
  end
```

Figure 4-14: Pseudo-code for Advancing Time

Throughout this process of splitting up particles, it is necessary to insure that each thread has all of the information required to calculate the forces on the particles for which it is responsible. This is accomplished through a bucket-based algorithm. When nodes are propagated from one thread to another, the algorithm divides the workspace into small buckets. As the number of buckets can be large, the simulator saves memory by storing the buckets in a hash table and only allocating those buckets that are actually used. The simulator places each single node in the bucket containing its position. It can then determine in constant time which nodes are close to those being propagated. Data on these close nodes, as well as those bound to the propagated nodes, is transmitted along with the propagated nodes. This guarantees that each processor has all data required to make the calculations assigned to it.

A thread-based algorithm should allow the simulator to make effective use of parallelism. The work required for individual nodes can vary considerably from one node to another, depending on how complicated each node is and how it interacts with its neighbors. Furthermore, the amount of work is hard to predict in advance; this makes the problem difficult to parallelize directly. However, it is well-addressed by Cilk's work-stealing system. When a thread requires very little work, the processor responsible for it should be freed to steal work from a more heavily burdened processor. The parallel run-time should still be linear in the number of nodes, as it requires linear time to transfer a subproblem to another processor. However, the constants should be improved. For an N node problem and p processors, if we assume the problem requires $k_1 N$ work serially, and that $k_2 N$ time is needed to send the work to another processor, the the total run-time in parallel should be approximately $k_1 \lceil \frac{N}{p} \rceil + k_2 N$. This should be an improvement if $k_1 >> k_2$ and $N >> p$.

4.4.3 Temperature/Brownian Motion

One persistent problem of numerical approximations to differential equations is that they introduce round-off errors that gradually degrade the quality of the approximation. For the most part, this is not a problem here, as it is not important that the path of a simulated particle be exactly what the physical model predicts over a long

period of time. However, these roundoff errors do introduce a serious problem in maintaining the amount of energy in the simulation. This quantity is important as it relates to the temperature of the solution; many biological problems are temperature sensitive, so maintaining a realistic temperature is a necessary feature of a simulation. In order to maintain simulation temperature in the presence of round-off error, the simulator relies on an adaptive method for simulating Brownian motion.

In order to explain this method, it is necessary to describe some issues of the relevant physics. One important point is that the temperature of a substance is related to the average kinetic energy of the particles it contains. Specifically, average kinetic energy per degree of freedom, $< KE >$, is given by the formula:
$< KE >= kT$
where k is Boltzmann's constant and T is the absolute temperature. Therefore, if a simulated solution is considered to be at a specific temperature, then the particles it contains should approximate this formula. The second issue to be addressed is Brownian motion. Brownian motion refers to seemingly random movements of particles in a solution. These movements result from collisions between the particles under observation and microscopic molecules in the solution.

The simulator applies a model of Brownian motion to the problem of maintaining the correct average kinetic energy. Under this method, when the forces on a particle from bonds and collisions are calculated, they are modified by two factors. The first is the damping factor, described in Section 4.1.3. The second is a component modeling Brownian motion, which has a random strength and direction. However, while the damping factor is user-specified, it is necessary for the program to choose the exact range over which random Brownian motion forces vary in order to insure convergence. The correct range of the random force is heavily dependent on the parameters of the simulation, the types of particles in use, and how they are interacting; this makes it infeasible to determine the range directly. The program handles this by selecting two multipliers, one for controlling the translational component of Brownian motion and another for controlling the rotational component, through an adaptive technique based on the deviation between actual and desired average kinetic energy. At the

beginning of each timestep, these two factors, *dscale* and *rdscale*, are modified based on Boltzmann's constant k, the temperature T, the timestep Δt, the average translational kinetic energy E_t, and the average rotational kinetic energy E_r, according to the following formulas:

$dscale = dscale(\frac{kT}{E_t})^{\Delta t}$

$rdscale = rdscale(\frac{kT}{E_r})^{\Delta t}$

The result is that the forces being applied increase when the average kinetic energy is lower than desired, causing kinetic energies to increase. Conversely, forces decrease when average kinetic energies are too high, allowing damping to dominate and resulting in a gradual decrease in kinetic energy.

This method provides a means of preserving long-term statistical accuracy while still obtaining short-term accuracy. Because random forces are relatively small and fluctuate rapidly, they have little effect on the position or velocity of a node over a short period of time. However, the adaptive system insures that kinetic energies will approximate statistically expected values. One side effect of this technique is that it is sometimes necessary to automatically decrease the timestep in order to avoid averaging out the random forces during the optimization procedure.

Chapter 5

Evaluation

This chapter evaluates how well the implementation meets the stated design requirements. It follows the format of Chapter 3, covering how well specific design criteria have been met. Section 5.1 examines whether the design has the necessary functionality. Section 5.2 evaluates the processing speed of the simulator on some typical problems. Section 5.3 evaluates how well the simulator meets the other constraints on its design.

5.1 Functionality

The simulator successfully implements the more realistic model of time and space required. It is capable of modeling free-floating particles in a solution. These particles can assemble in any order and at any rate. Furthermore, the simulator allows time to be advanced by a quantifiable amount, rather than to a state of energy relaxation.

The simulator also supports a more realistic kinetic model of binding. Edges bond probabilistically according to user-specified bonding energies. The probabilities of bonding are related to a realistic model of binding kinetics. The actual forces exerted by bonds are less realistic, as they use a fairly simple spring model; however, they appear adequate for the purposes of the simulator.

The simulator also implements a more physically reasonable model of coat proteins. Users can set the mass and size of a node and can create different shapes

through unions of spheres. Furthermore, nodes support arbitrary bond configurations. The simulator also supports conformational switching of both entire nodes and particular domains within nodes. One caveat is that specifying shapes as unions of spheres, while in principle very versatile, in practice requires considerable work when generating more complicated shapes.

Additionally, the design makes the simulator very versatile. Users have control over many parameters of the simulation. The use of a high-level simulator control language allows reconfiguration of many aspects of simulator behavior, as the alternate controller of Section 4.2.5 demonstrates. Many other aspects of the design capitalize on this, such as the use of generalized programs for data files and the use of programmable hooks to link the interface to the command interpreter. There are, however, some areas in which the simulator is not easily configured. Beyond some specific parameters, the numerical routines cannot be modified by users; therefore users are limited to a specific representation of proteins and model of the laws of physics. Furthermore, the serial/parallel interface consists of specific hard-coded routines, so users cannot extend it to issue new commands to the numerical simulator or to request data the numerical simulator does not currently provide.

Finally, the simulator generally meets the requirement of ease of use. The graphical interface makes it easy to control most aspects of a simulation. Users can enter most common commands through a simple point-and-click interface. Furthermore, it is fairly easy to design nodes and edges. Because parameters are designed to use physically realistic values, it is generally not difficult to select valid parameters. Finally, the use of graphical output makes it simple to see and understand how a simulation is progressing. The only area in which ease of use is not well supported is when direct access to the simulator control language is required. To some degree, this is a necessary tradeoff between simplicity and versatility. However, programming in the control language is more difficult than it might be, since it lacks debugging support and the syntactic sugar typical of Lisp variants.

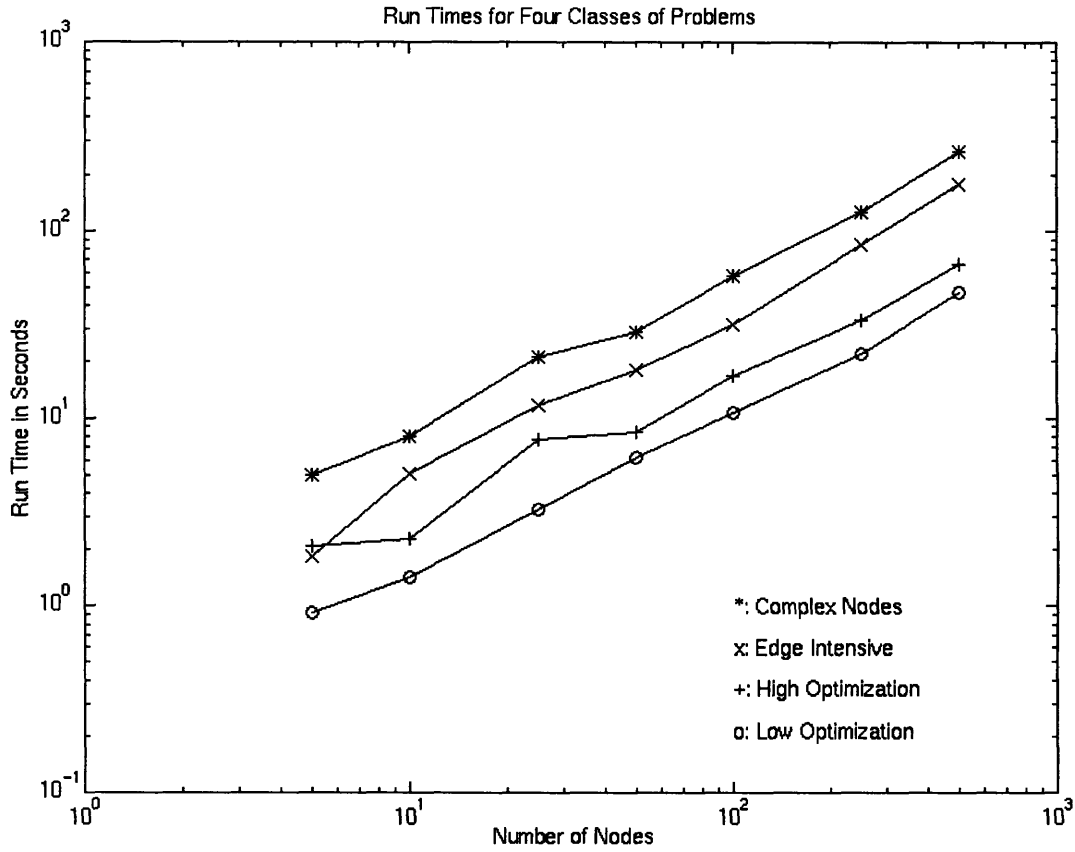


Figure 5-1: Results of Performance Tests on Four Problems

5.2 Performance

Overall, processing speed is sufficient to make the simulator practical, but is less than desirable. Performance in advancing the simulation in time was measured for a range of different numbers of nodes and for four different kinds of problem: simple nodes and low optimization, simple nodes and high optimization, edge intensive, and complicated nodes. The results of these tests are shown in Figure 5-1. The graph is a log-log plot of the time required to run ten timesteps for each of the four types of problem, with 5, 10, 25, 50, 100, 250, and 500 nodes. These were run on a CM-5 using thirty-two processors.

One implication of the graphs is that while the code is exploiting parallelism, it is not using it very effectively. The slopes of the graphs are only slightly less

than one, suggesting limited use of parallelism. This in part reflects the fact that not all of the computationally expensive code is parallelized. It may further result from the use of variable timesteps; this strategy could cause the simulator to spend a large percentage of its time on a small proportion of the nodes, thereby making it unable to exploit parallelism well except on very large problems. However, the results also suggest that the cost of transferring a sub-problem from one processor to another may be too high, minimizing the benefits of transferring problems. Of the four classes of problems, all seem to make about the same use of parallelism except for the edge intensive case, which scales more poorly than the others. This is to be expected, since edge connections increase the amount of data that must be communicated between processors.

In an absolute sense, however, the results are sufficient for real simulations. In the worst case, the time required for ten timesteps was 267.514 seconds, or approximately 26.7514 seconds per timestep. While this is much slower than desirable, it adequate for running a simulation for a few thousand timesteps in a reasonable time; this is sufficient for many kinds of simulation problems. Furthermore, the low optimization test is approximately an order of magnitude faster, and is likely to more accurately reflect realistic performance. This time is still much higher than ideal, but should be sufficient for currently planned tests. Furthermore, the high optimization tests, which reduce errors to one-tenth that in the low optimization tests, have an additional cost of approximately a factor of two. This suggests that performance need not unduly be sacrificed to assure accuracy when it is needed.

The simulator does manage to avoid other potential performance problems. Graphics, while slower than ideal for an interactive simulation, are fast enough not to become a problem, particularly when the user interface runs on an SGI. Furthermore, the serial/parallel interface allows a graphical simulation while keeping network bandwidth and memory usage on the CM-5 relatively low.

The overall conclusion is that the simulator meets its design goals in this area, although there is considerable room for improvement. The simulator is fast enough to run the required tests for reasonable simulations. However, it is much slower than

desirable, which can be expected to delay work with the simulator. Furthermore, this may make it difficult to apply the simulator to more complicated problems in the future. Therefore, while its performance is adequate, it should be improved.

5.3 Other Constraints

The simulator makes good use of the available hardware. Because of the serial/parallel split and the use of Cilk in the numerical simulator, it is able to take advantage of the available CM-5. However, due to Cilk's portability, it can also run on several other architectures. Likewise, the interface is portable to different platforms. The high-quality OpenGL graphics have only been run on an SGI, but, with Vogle's lower-quality graphics, the user interface can be ported to other Unix workstations. The interface does, however, use the X window system and Unix network sockets, so it could not easily be ported to non-Unix machines.

Lastly, the development time of the simulator is within the required limits. While work is continuing, the simulator is currently functional and is being applied to real problems in virus shell assembly. Furthermore, a functional simulator has been produced early enough to allow time for testing and documentation. Therefore, the scope of the project seems to have been sufficiently limited to allow its completion in a timely manner, without being simplified so much that it ceased to be useful.

Chapter 6

Applications

Although the central focus of this thesis is the development of the simulator itself, two sample applications are described. These examples help to demonstrate what kinds of problem the simulator can answer, provide an indication of its versatility, and show how it works in practice. Section 6.1 describes the results of attempting a kinetic model using undirected assembly. Section 6.2 examines a use of the alternate controller described in Section 4.2.5 as a means to experiment with feasible shapes of nodes.

6.1 Undirected Assembly

An important function of the simulator is exploring possible kinetic models for implementing local rules. One possible model explored by the simulator is that implicitly used by the prior simulator: undirected assembly with high binding energies. Undirected assembly means that protein subunits are not predisposed towards assembling a shell in any particular order; whether or not a particle has already formed a bond on one edge does not affect its probability of forming a bond on any other edge. High binding energy means that compatible edges have a high probability of bonding if they occur in the appropriate relative positions and a low probability of breaking a bond once it has formed. In order to test the feasibility of this model, the simulator was applied to a set of rules implementing high binding energies and undirected

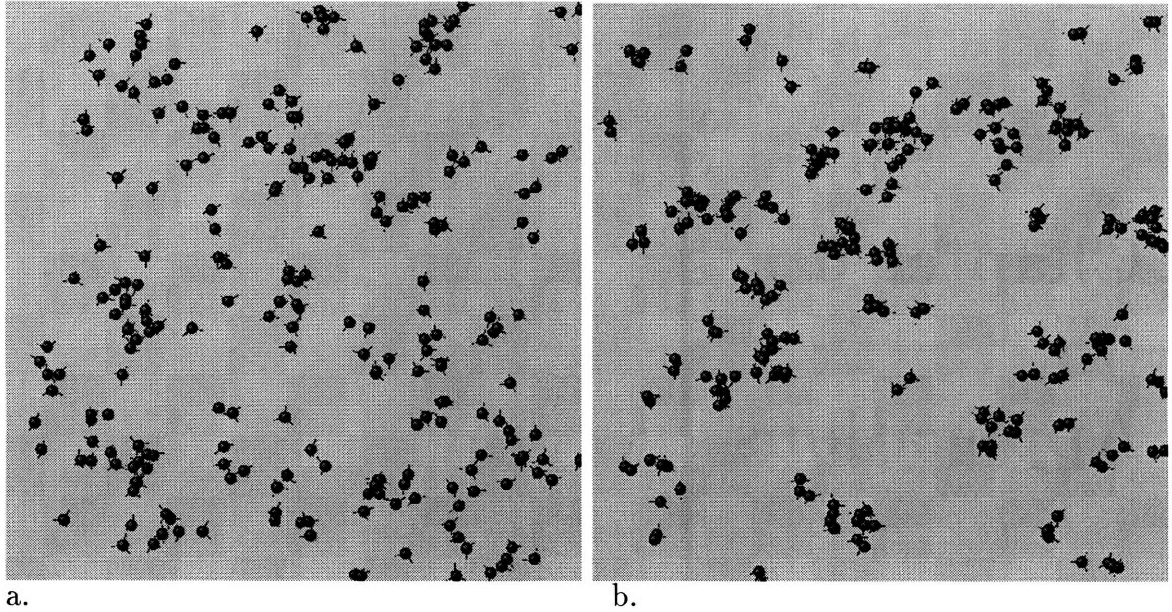


Figure 6-1: Results of Undirected Assembly at Two Points in Time

assembly.

The simulation attempted to model assembly of a T=1 shell. A T=1 provides a good preliminary test of the model, as it is the simplest shell. This test attempted to imitate the abstract model of coat proteins used by the prior simulator. The rules used modeled the shape of a protein subunit with a single sphere. Bonding angles were chosen to be ideal for the T=1 lattice. The sizes and shapes of the coat proteins were not, however, chosen to model actual coat proteins. All binding energies were set at -15 kcal/mol, which provides a very a high probability of forming and maintaining bonds.

The result of the simulation was that subunits quickly formed small clusters, but could not easily combine further. Figure 6-1a shows the simulation after 40 time-steps. Several small aggregates are visible, showing that individual particles can combine without much difficulty. However, very little further progress occurs even after a long time. Figure 6-1b shows the same simulation after 400 time-steps. While almost all particles have joined small aggregates, larger ones have not formed. Apparently, once particles form into small clusters it becomes much more difficult for them to combine further.

These results suggest that the proposed model is not adequate for actual shell assembly. Given the problems observed, it appears that shell assembly would occur more efficiently if individual subunits could easily attach to growing shells, but not to other free-floating subunits. This suggests that it may be necessary to constrain pathways of growth in order to allow efficient shell assembly.

This application helps to show some of the utility of the simulator. The questions of if and how assembly is constrained are important to understanding shell growth and cannot easily be examined experimentally. While theoretical work offers some insight into the problem, the prior simulator cannot model its relevant features. The new simulator, because of its more realistic model of kinetics and its ability to simulate free-floating particles, could provide some additional answers.

6.2 Modeling with the Alternate Controller

A very different application uses the alternate controller described in Section 4.2.5 to experiment with possible shapes for coat proteins. It is not clear how important the shapes of coat proteins are to assembly, or how they affect it. The simulator provides one possible means of exploring this question. However, because of the many parameters involved in developing a good model for a coat protein, it can be difficult to fine-tune any particular parameter in a general simulation. The alternate controller offers a way around this, by providing a middle ground between the old simulator and the full capabilities of the new simulator. With the alternate controller, nodes can be added one at a time to a growing shell, thereby avoiding delays of waiting for the shell to assemble, but with orders of assembly determined by binding energies. By using a shell with high binding constants and well-optimized angles, the alternate controller can produce a simulation in which it is possible to experiment with feasible node shapes without interference from other factors.

The use of a correct protein shape may be important in generating valid simulator results. The earlier simulator used simple spherical masses for all proteins. While this is adequate for a more abstract model, it is not suitable for understanding the

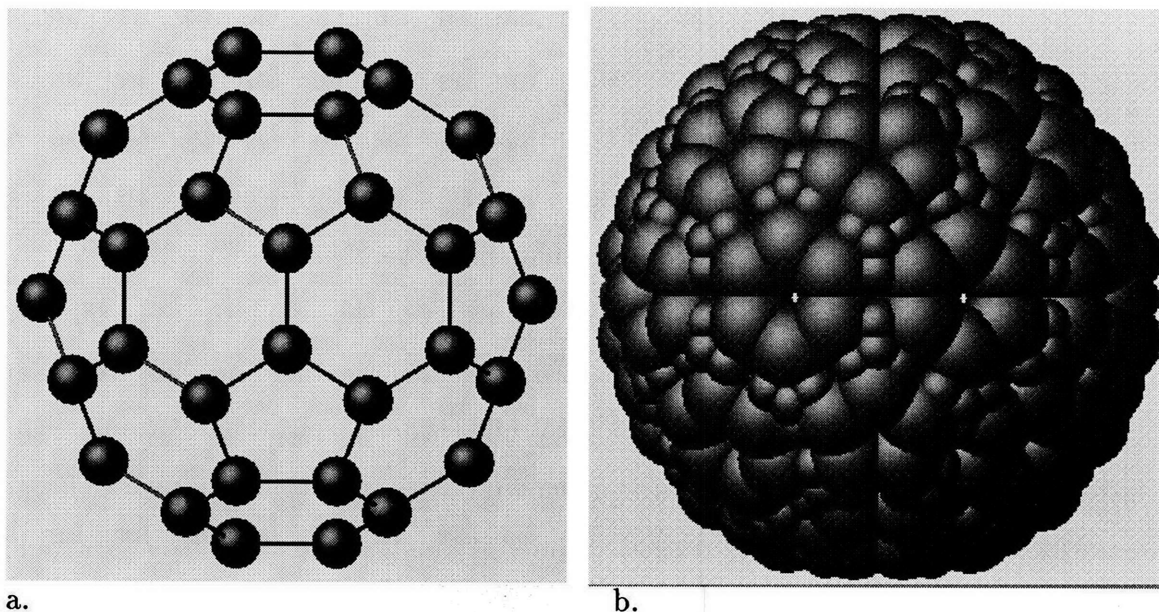


Figure 6-2: T=1 Shells Produced by Simple and Refined Models

effects of node shape on assembly. For example, the packing of proteins against each other may be crucial to stabilizing the shell or determining its size.

For this application, simulations were attempted on a T=1 shell. The result of the first attempt, with a simple model involving a single sphere, is shown in Figure 6-2a. This model looks similar to that of the previous simulator. Several attempts were made to refine this into a more realistic, space-filling model. Repeated refinements resulted in the shell of Figure 6-2b. This shell uses complex nodes, consisting of seven spheres each, to create approximately triangular coat proteins. These proteins fill most of the empty space of the surface of the shell. The result is a more rigid shell, as proteins have less freedom to move around within it.

This application further helps to demonstrate the utility of the new simulator. It shows the usefulness of the alternate controller in experimenting with node parameters. This, in turn, demonstrates the value of designing a simulator sufficiently versatile to support the alternate controller. It also shows that the model of complex nodes implemented does give users considerable latitude for designing simulated nodes that realistically model actual coat proteins.

Chapter 7

Discussion

This section examines some lessons of developing the simulator and discusses ideas for the future. Section 7.1 summarizes the information presented and draws some conclusions about the project, as well as broader work in the field. Section 7.2 describes some avenues for future work on the project.

7.1 Conclusions

This paper has described the construction of a simulator for realistically modeling the reaction kinetics of virus shell assembly. The simulator successfully achieves the design goals set out for it. It provides physically reasonable models of the relevant features of coat protein interactions. It achieves this quickly enough to make it useful without unduly taxing computational resources, and it provides a versatile and easy to use interface.

The simulator extends the range of phenomena that can be simulated and questions that can be answered regarding virus shell assembly. It offers physically realistic models of viral coat proteins, the kinetics of their interactions, and the environment in which they form. This provides a means of examining aspects of shell assembly that cannot currently be observed in the laboratory and were beyond the scope of prior simulation work. It should, therefore, be a valuable tool in future work with the local rules theory of virus shell assembly.

One lesson of the project regards the role of simulation work in a larger study in biology. The project shows that computer simulation can provide a means to observe physical processes that current laboratory techniques cannot analyze. Also, once the initial effort of constructing a simulator is finished, it can often perform experiments more easily and less expensively than comparable laboratory work. As algorithms and hardware improve, simulation is likely to become a more important tool for understanding biochemistry. However, it is also important to remember the limitations of simulator work. A computer simulation must rely on assumptions, abstractions, and approximations. Thus, in interpreting the results of this or any simulator, it may be difficult to sort out what data accurately reflects the underlying physics and what is an artifact of the simulator. A simulator is valuable as a verification of concept for theoretical work, as a tool for determining what avenues of exploration are likely to be most valuable, or as a means of generating hypotheses about the systems it simulates. Its output, though, should be regarded as theories to be tested in a laboratory, not as complete results in themselves. Simulators such as the one described here are potentially crucial tools for future work in biology, as long as their limits are acknowledged.

Perhaps the most important lesson of experience with this simulator is that there is a place for mid-level simulators like it that combine realism where it is needed with abstraction elsewhere. The simulator provides physically reasonable models of reaction kinetics and thermodynamics and the ability to modify many parameter of simulated proteins and their environment; other aspects of the simulator, such as the use of springs to model binding, are more abstract. This combination makes it possible to gather data that could not be obtained with a more abstract model, such as the older virus shell simulator, while allowing simulation of problems that would be computationally infeasible for a lower-level simulator, such as one that models coat proteins at the atomic level. This suggests that exploring problems at the edge of what is computationally tractable may create a permanent need for the design of custom simulators that are capable of realistically modeling what is needed for a specific application, but abstracting away what is not. It is likely that increasing

computer processing speeds will only amplify this trend, as a greater range of problems approach the point of being computationally feasible.

7.2 Future Work

Future work with this simulator will fall into two general categories: improvements to the simulator itself, and additional applications of the simulator. Section 7.2.1 describes some areas in which the simulator might be improved and conjectures what impact these improvements would have. Section 7.2.2 describes extensions of the applications already described, other planned applications, and other areas in which the simulator might be useful.

7.2.1 Potential Improvements to the Simulator

One potentially valuable area for future work would be to explore methods of improving the performance of the simulator. When high accuracy is required, the simulator may need to perform a considerable amount of work to converge on a valid solution; switching to a numerical method that requires less work could therefore greatly improve performance. It is not obvious how this might be accomplished, however, as there are tradeoffs in all common strategies. It would be worthwhile to examine other schemes and evaluate how they might affect the performance of the simulator.

Another area in which performance might be improved is through the use of global memory. Since this project was begun, the developers of Cilk have added support for a global memory abstraction[5]. There are some areas of the simulator code that would almost certainly benefit from the use of global memory. These would include definitions of node and edge types, global parameters such as the timestep and temperature, and certain information about individual nodes which must be identical from processor to processor and which is changed only infrequently. In other areas, the value of global memory is more questionable. It might be worthwhile as a replacement for explicit data propagation in the procedure for advancing time; however, this could also decrease performance by causing unnecessary data transfers.

Whether global memory would be a net advantage requires further consideration.

It would also be valuable to explore ways of improving the degree of parallelism in the program. The performance results in Section 5.2 suggest that the simulator is not making as efficient use of parallelism as it might. A more nearly optimal parallelization on the currently available hardware might improve performance by an order of magnitude or more.

The user interface could also be improved to allow easier use. Currently, most common aspects of the simulator's use should be easy to learn, as a user generally only needs to deal with the graphical interface. One exception is in developing rules for a simulation. The simulator cannot be used without defining the necessary node and edge types and simulation parameters. However, choosing these and coding them in a file is not always easy. Remedying this is therefore a potentially important consideration in future work with the simulator. One possible approach is to create an external utility to assist in designing nodes and choosing their parameters. Ease of use could also be improved by making the control language cleaner and adding syntactic sugar to simplify some common operations.

7.2.2 Other Applications

One planned application is to extend the work described in Section 6.1 on pathways of assembly. The simulator will be used to explore different orders in which assembly might proceed. It will also examine kinetic mechanisms by which assembly might be constrained. It will additionally explore the effects of different assembly models on the rate of shell growth.

The simulator is also meant to explore other physical properties of coat proteins. This includes extending the work described in Section 6.2 to explore in more detail how the shapes of subunits affect shell assembly, particularly for larger shells. Furthermore, the simulator will be used to examine different kinetic models for how coat proteins might attach to each other. Finally, it will be used to look at the significance of different models of how and when conformational switching occurs. In particular, it should be possible to look at complicated rule sets, such as the four conformation

T=7 rules described in Section 2.2.

Beyond these currently anticipated applications, it may be possible to apply the simulator to problems other than icosahedral virus shell assembly. It may be able to simulate some non-icosahedral viruses. It could also examine applications of local rules to other self-assembly systems in nature. Finally, the simulator may provide a means of exploring possible designs for artificial self-assembly systems.

Bibliography

[1] T. S. Baker, D. L. D. Caspar, and W. T. Murakami. Polyoma virus "hexamer" tubes consist of paired pentamers. *Nature*, 303:446–448, 1983.

[2] B. Berger and P. W. Shor. Local rule switching mechanism for viral shell geometry. MIT Laboratory for Computer Science Technical Memo 527, 1995.

[3] B. Berger, P. W. Shor, L. Tucker-Kellog, and J. King. Local rule-based theory of virus shell assembly. *Proc. Natl. Acad. Sci. U.S.A.*, 91:7732–7736, 1994.

[4] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[5] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *International Parallel Processing Symposium*, April 1996.

[6] S. Casjens and J. King. Virus assembly. *Annu. Rev. Biochem.*, 44:555–611, 1975.

[7] D. L. D. Caspar and A. Klug. Physical principles in the construction of regular viruses. *Cold Spring Harbor Symp. Quant. Biol.*, 27:1–24, 1962.

[8] T. Dokland, B. H. Lindqvist, and S. D. Fuller. Image reconstruction from cryo-electron micrographs reveals the morphopoietic mechanism in the P2-P4 bacteriophage system. *EMBO J.*, 11:839–846, 1992.

[9] W. Earnshaw and J. King. Structure of phage P22 coat protein aggregates formed in the absence of the scaffolding protein. *J. Mol. Biol.*, 126:721–747, 1978.

[10] E. H. Echidna. Vogle 1.3.0. Available from **wuarchive.wustl.edu** via anonymous ftp in directory **mirrors/echidna/src**, 1993.

[11] B. N. Fields and D. M. Knipe, editors. *Fields Virology.* Raven Press, New York, second edition, 1990.

[12] H. Fraenkel-Conrat and R. C. Williams. Reconstitution of active tobacco mosaic virus from its inactive protein and nucleic acid components. *Proc. Natl. Acad. Sci. U.S.A.*, 41:690–698, 1955.

[13] S. C. Harrison, A. J. Olson, C. E. Schutt, F. K. Winkler, and G. Bricogne. Tomato bushy stunt virus at 2.9 Å resolution. *Nature*, 276:368–373, 1978.

[14] I. Katsura. Structure and inherent properties of the bacteriophage lambda and head shell iv: Small-head mutants. *J. Mol. Biol.*, 171:297–317, 1983.

[15] R. C. Liddington, Y. Yan, J. Moulai, R. Sahli, T. L. Benjamin, and S. C. Harrison. Structure of simian virus 40 at 3.8-Å resolution. *Nature*, 354:278–284, 1991.

[16] O. J. Marvik, T. Dokland, R. H. Nøkling, E. Jacobsen, T. Larsen, and B. H. Lindqvist. The capsid size-determining protein sid forms an external scaffold on phage P4 procapsids. *J. Mol. Biol.*, 251:59–75, 1995.

[17] D. Muir. Simulated construction of viral protein shells. Bachelor's Thesis. Massachusetts Institute of Technology, 1994.

[18] B. V. V. Prasad, P. E. Prevelige, Jr., E. Marietta, R. O. Chen, D. Thomas, J. King, and W. Chiu. Three-dimensional transformation of capsids associated with genome packing in a bacterial virus. *J. Mol. Biol.*, 202:824–835, 1993.

[19] I. Rayment, T. S. Baker, and D. L. D. Caspar. Polyoma virus capsid structure at 22.5Å resolution. *Nature*, 295:110–115, 1982.

[20] D. Rogelberg and J. C. Fullagar, editors. *OpenGL Reference Manual.* Silicon Graphics, Inc., 1995.

[21] M. G. Rossmann. Constraints on the assembly of spherical virus particles. *Virology*, 134:1–13, 1984.

[22] H. G. Schlegel. *General Microbiology.* Cambridge University Press, 1993.

[23] P. A. Thuman-Commike, B. Greene, J. Jakana, B. V. V. Prasad, J. King, P. E. Prevelige, Jr., and W. Chiu. Three-dimensional structure of scaffolding-containing phage P22 procapsids by electron cryo-microscopy. *J. Mol. Biol.*, 1996. in press.

Appendix A

Specifications for the Control Language

This appendix describes the simulator control language in detail. Section A.1 examines some general features of the language. Section A.2 describes how the control language connects to the graphical interface. Section A.3 presents specifications for the primitive operations and special forms supported by the control language.

A.1 General Features

The control language is typical in many ways of other Lisp-like languages. The language uses dynamic binding. It also employs eager evaluation, with the exception of the evaluation of certain special forms, described in Section A.3.1. The language contains seven basic data types, two compound data types, two data types for procedure abstractions, and two special data types. Deallocation of memory is handled automatically through a mark and sweep garbage collector.

The general syntax of the control language consists of two basic components: atoms and procedure evaluations. An atom can be any of the data types described in the remainder of this section. A procedure application is of the form *(operator [operand]*)*; here, each term may be an atom or a procedure evaluation. An operand can evaluate to any value, but the operator must either be a special form or evaluate

to a procedure object, which may be either a primitive procedure or a user-defined procedure.

The control language contains seven basic data types. Label values consist of strings of characters and are used to name objects of other types. Numeric values are used for numbers. They are internally represented by floating point values, but are used for both integer and floating point arithmetic. Boolean values can be either true or false. String values consist of null-terminated sequences of characters, and are specified by enclosing the characters in quotation marks. Bitfield values consist of sequences of bits. Fileid values are pointers to files that can be used to read or write strings of data to or from a file.

In addition to these basic types are two compound types: list and array. A list consists of zero or more values, each joined to the next. Elements of a list can be of any type, including list, except for the two special types described below. Lists are usually accessed through two special forms, first and rest, that access the first element of the list and the remainder of the list, respectively. Lists are generally useful when creating structures whose size changes, as an additional element can be added to a list in constant time. Arrays also consist of ordered sets of values, but have fixed sizes specified when they are created. Elements can be examined or inserted into an array in constant time. When an array is created, all of its elements are initially inaccessible and must be explicitly added before they can be accessed without returning an error.

There are also two data types for procedure abstractions: primitive and non-primitive procedures. Primitive procedures are hard-coded into the interpreter and cannot be created by users. When the interpreter first starts up, each primitive procedure is bound to a label, although it is possible to bind the primitive procedures to other labels or to bind their labels to other values. Although most primitive procedures were created to add functionality to the language, some were created for efficiency reasons. Non-primitive procedures are defined by the user or through data files. They contain a possibly null list of arguments and a body evaluated at run time, substituting supplied values for the arguments.

Finally, there are two special types: none and error. Values of type none are used

only as place holders in uninitialized arrays, and cannot be accessed by users. Type error requires a more detailed explanation. When primitive procedures or most special forms encounter errors, such as improperly formatted argument lists, they returns values of type error, which have associated with them strings explaining the errors. Errors are typically propagated up through an expression, so that if one argument to a procedure is an error, then the value of the procedure application will be the same error. Error values cannot in general be treated like other data types. It is not possible to bind a label to an error, or to insert an error into a list or array. However, users can utilize the error propagation system through two special forms: make_error and errortrap. Make_error creates a user-defined error, while errortrap tests if its argument is an error value and, if so, stops it from propagating. Make_error and errortrap are described in more detail in Section A.3.1.

A.2 Hooks to the Graphical Interface

The graphical interface connects to the simulator control language through several "hooks," procedures in the control language called to activate specific functions. These hooks control the response to user actions and certain aspects of updating the display and communicating with the numerical simulator. This section describes the hooks, the conditions under which they are activated, and their default behaviors.

Seven of the hooks control the graphics display. Procedure zoom_in is activated by pressing the "zoom_in" button and procedure zoom_out by pressing the "zoom_out" button on the graphical interface. Procedures rotate_x, rotate_y, and rotate_z, are activated by the "rotate" button. Each takes one numerical argument supplied by the user when the "rotate" button is used. Procedure recenter, activated by the "re-center" button, takes three numerical arguments provided by the user, representing a point in the workspace. Finally, procedure draw_workspace is activated when a user calls any of the other graphics commands or updates the simulator time, or when the graphics window is partially obscured then uncovered. Default specifications for these procedures are as follows:

zoom_in = **procedure** () **returns** boolean
effects: zoom_in multiplies the distance between the virtual eye point and the focal point by .8. It returns true.

zoom_out = **procedure** () **returns** boolean
effects: zoom_out multiplies the distance between the virtual eye point and the focal point by 1.25. It returns true.

rotate_x = **procedure** (numeric ang) **returns** boolean
effects: rotate_x rotates the eye point counter-clockwise around the X-axis by ang degrees relative to the focal point. It returns true.

rotate_y = **procedure** (numeric ang) **returns** boolean
effects: rotate_y rotates the eye point counter-clockwise around the Y-axis by ang degrees relative to the focal point. It returns true.

rotate_z = **procedure** (numeric ang) **returns** boolean
effects: rotate_z rotates the eye point counter-clockwise around the Z-axis by ang degrees relative to the focal point. It returns true.

recenter = **procedure** (numeric x, numeric y, numeric z) **returns** boolean
effects: recenter sets the focal point to (x,y,z). It returns true.

draw_workspace = **procedure** () **returns** boolean
effects: draw_workspace draws all visible nodes and edges in accordance with the current focal point and eye point. It returns true.

Two hooks are used to handle queries about the status of nodes. Both are called through the "query" dialog box, activated by pressing the "query" button on the graphical interface. Procedure process_node_query is activated by choosing to specify nodes by number in the "query" dialog box, and is passed a numeric value supplied by the user in the number field of that dialog box. Procedure query_region is called by specifying nodes by region in the "query" dialog box, and is passed six numeric values, supplied by the user in the xmin, xmax, ymin, ymax, zmin, and zmax fields of that dialog box. In each case, the output of the command is displayed in the feedback window of the graphical interface. Specifications for these two commands are as follows:

process_node_query = **procedure** (numeric n) **returns** string
effects: process_node_query queries the numerical simulator for the position, velocity, angular rotation, and angular velocity of node n and returns a string containing the results in the form:
"Pos: *position*
Vel: *velocity*
Ang: *angular rotation*
AngVel: *angular velocity*".

query_region = **procedure** (numeric xmin, numeric xmax, numeric ymin, numeric ymax, numeric zmin, numeric zmax) **returns** string

effects: query_region queries the numerical simulator for the positions, velocities, angular rotations, and angular velocities of each node with x coordinate between xmin and xmax, y coordinate between ymin and ymax, and z coordinate between zmin and zmax. It averages all values for each of the four fields and returns a string containing the results in the form:
"Average values:
Pos: *average position*
Vel: *average velocity*
Ang: *average angular rotation*
AngVel: *average angular velocity*".

Four more hooks are used for adding and deleting nodes. Procedures add_node and add_random_node are activated through the "add" dialog box. If the "random" option is selected in that box, then add_random_node is called one or more times, depending on how many nodes the user has chosen to add, and passed a list of node types which it uses to randomly select the type of node to add. If the "random" option is not selected, then add_node is called and passed user-specified position, velocity, angular rotation, and angular velocity values; in this case, only one node is created, with the arguments supplying its state. Procedures delete_node and delete_region are activated through the "delete" dialog box. If the "by number" option is chosen, then the user specifies a node index which is passed to delete_node, instructing it to delete the node with the given index. If the "by region" option is selected, then the user must supply minimum and maximum X, Y, and Z values, which define a region within which all nodes are to be deleted. The default specifications for these procedures are as follows:

add_node = **procedure** (numeric t, list p, list v, list r, list a) **returns** list
modifies: the_nodes, num_nodes, the_edges, num_edges, the_singles, num_singles, num_variables
effects: add_node adds a node with type t, position p, velocity v, angular rotation r, and angular velocity a. p, v, r, and a are expressed as lists of three numerical values each. add_node adds the new node to the list of nodes and its edges and single sub-nodes to the appropriate lists and increments counters for the number of nodes, edges, singles, and variables. It returns a list of the node parameters followed by the newly defined node.

add_random_node = **procedure** (list n) **returns** list
modifies: the_nodes, num_nodes, the_edges, num_edges, the_singles, num_singles, num_variables
effects: add_random_node behaves just as add_node does, except that the added node has a random type chosen from a list of node types in n, its position is randomly chosen within the workspace, its angular rotation is random and it has a random velocity and angular rotation of magnitude between 0 and 1 in each component.

delete_node = **procedure** (numeric n) **returns** numeric

modifies: the_nodes, num_nodes, the_edges, num_edges, the_singles, num_singles, num_variables
effects: delete_node deletes a node from the simulation, removing the node, its edges, and its singles from the relevant arrays and decrementing the counters of nodes, edges, singles, and variables. It returns the number of nodes present prior to the deletion.

delete_region = **procedure** (numeric xmin, numeric xmax, numeric ymin, numeric ymax, numeric zmin, numeric zmax) **returns** boolean
modifies: the_nodes, num_nodes, the_edges, num_edges, the_singles, num_singles, num_variables
effects: delete_region behaves as does delete_node, except that it deletes all nodes with position (x,y,z) such that xmin≤x≤xmax, ymin≤y≤ymax, and zmin≤z≤zmax. It returns false.

An additional four hooks are used to define node and edge types. The four hooks are add_edge_type, add_single_node_type, add_complex_node_type, and add_variable_node_type. Their purposes are self-explanatory. These hooks are not directly accessible through the graphical interface and are generally used within a rules file to set up the parameters of a simulation. They have the following default specifications:

add_edge_type = **procedure** (list o, numeric ks, numeric kt, numeric kr, numeric er, numeric ed, numeric r, numeric a) **returns** string
modifies: the_edge_types, num_edge_types
effects: add_edge_type creates an edge type capable of connecting to edges listed in o; with spring constants ks, kt, and kr; energies er and ed; distance tolerance r; and angle tolerance a. It returns a string signifying that it has finished.

add_single_node_type = **procedure** (numeric r, numeric c, numeric m, list poses, list ups, list edges) **returns** string
modifies: the_node_types, num_node_types
effects: add_single_node_type creates a single node type with parameters as specified in the argument list and adds this type to the final position in the_node_types, incrementing num_node_types. It returns a string signifying that it has finished.

add_complex_node_type = **procedure** (list pos, list ang, list nodes) **returns** string
modifies: the_node_types, num_node_types
effects: add_complex_node_type creates a complex node type with parameters as specified in the argument list and adds this type to the final position in the_node_types, incrementing num_node_types. It returns a string signifying that it has finished.

add_variable_node_type = **procedure** (list energy, list nodes) **returns** string
modifies: the_node_types, num_node_types
effects: add_variable_node_type creates a variable node type with parameters as specified in the argument list and adds this type to the final position in the_node_types, incrementing num_node_types. It returns a string signifying that it has finished.

There are five additional hooks used for other general tasks of the simulator. Procedure restart, activated by the "restart" button, reinitializes the simulator data.

Procedure do_step_forward, activated by the "step" button, advances the simulator time. Procedure resize_workspace sets the dimensions of the current workspace; like the commands for defining node and edge types, it is not directly accessible from the graphical interface and is meant to be used in rules files when defining the parameters of a simulation. Finally, procedure do_save, activated by pressing the "save" button, saves the state of the current simulation. It is passed a user-supplied file name to use for the save file. The default specifications of these procedures are as follows:

restart = **procedure** () **returns** boolean
  modifies: anything
  effects: restart sets simulation parameters to their initial values, clears the display, closes the connection to the numerical simulator, if any, and attempts to establish a new connection at the machine and port specified by the global variables host and port. It returns false.

do_step_forward = **procedure** () **returns** string
  modifies: workspace_data
  effects: do_step_forward instructs the numerical simulator to advance the time updates_per_run·steps_per_update steps. do_step_forward updates the display after each steps_per_update steps and leaves updated values of the current data in the global variable workspace_data. It returns the string "Stepped forward".

resize_workspace = **procedure** (numeric x, numeric y, numeric z) **returns** boolean
  modifies: workspace_width, workspace_height, workspace_depth, graphics_eye
  effects: resize_workspace sets workspace_width, workspace_height, and workspace_depth to x, y, and z respectively. It moves graphics_eye, the eye point of the graphics display, so that the direction from it to the focal point is unchanged, but the distance is 1.44 times the maximum of x, y, and z. It returns false.

do_save = **procedure** (string f) **returns** fileid
  effects: do_save attempts to open a file with name f, write out a save file capable of reconstructing the current state of the numerical simulator, close the file, and return its fileid. If it is unable to open a file with the specified name for writing, it returns an error.

A.3 Primitive Routines and Special Forms

This section provides specifications for the special forms and primitive functions implemented by the simulator. In all specifications, the argument list given represents a correctly formed argument list. If the argument list is not correctly formed, the procedure will return an error value regardless of its stated return type. Type "any" in an argument list means that any type can be passed as an argument, while as a

return value it means that any type could be returned. Section A.3.1 provides specifications for special forms. Section A.3.2 gives specifications for general primitive procedures. The remaining three sections provide specifications for three specialized subsets of the primitive procedures. Section A.3.3 gives specifications for graphics routines, Section A.3.4 for routines for communicating with the numerical simulator, and Section A.3.5 for routines for vector math.

A.3.1 Special Forms

Special forms are commands that are not stored as primitive procedures. Therefore, they cannot be bound to any other labels. Typically, this is done because they alter the usual rules for evaluating an expression. For instance, in some cases the arguments to a special form are not evaluated before being passed to it, whereas for primitive procedures, arguments are always evaluated first. The specifications for the special forms are as follows:

define = **procedure** (label l, any a) **returns** any
    modifies: l
    effects: define binds l to a and returns the value of a.

errortrap = **procedure** (any a) **returns** list
    effects: errortrap returns a list of two items. If a evaluates to an error then the first item is the boolean true and the second is a's associated string. Otherwise, the first item is the boolean false and the second is the value of a.

first = **procedure** (list l) **returns** any
    effects: first returns the first element of l, if any, or an error if l is empty.

get_array = **procedure** (array a, numeric n) **returns** any
    effects: If 0<=n< |a| and element n of a, indexed from zero, has been initialized, get_array returns the value of that element. Otherwise, it returns an error.

if = **procedure** (boolean b, any e1, any e2) **returns** any
    effects: if evaluates b and returns the value of e1 if b is true, e2 if b is false. if does not evaluate whichever of e1 and e2 is not returned. e2 is optional and if it is not included and b is false then if returns false.

make_array = **procedure** (numeric n) **returns** array
    effects: If n>0, make_array returns an empty array of length n. Otherwise, it returns an error.

make_error = **procedure** (string s) **returns** error
    effects: make_error returns an error with associated string s.

make_list = **procedure** (any a1, any a2, ...) **returns** list
    effects: make_list returns a list containing the values of its arguments, in order.

procedure = **procedure** (list l, any a) **returns** procedure
effects: procedure returns a procedure with variables listed in l that evaluates to the value of a with its arguments substituted for any occurrences of the variables in l.

quote = **procedure** (any a) **returns** any
effects: quote returns a unevaluated.

rest = **procedure** (list l) **returns** list
effects: rest returns all of l except the first element if l is not empty. Otherwise, it returns an error.

sequence = **procedure** (any a1, any a2, ...) **returns** any
effects: sequence evaluates each argument in order, returning the value of the last argument.

set = **procedure** (label l, any a) **returns** any
modifies: l
effects: If l is bound to a value, set replaces that value with the value of a and returns the old value. Otherwise, it returns an error.

set_array = **procedure** (array a, numeric n, any x) **returns** array
modifies: a
effects: If $0 \leq n < |a|$, set_array sets element n of a, indexed from zero, to the value of x and returns the modified array.

set_first = **procedure** (list l, any a) **returns** any
modifies: l
effects: set_first sets the first element of l to the value of a and returns the replaced value.

set_rest = **procedure** (list l1, list l2) **returns** list
modifies: l1
effects: set_rest replaces all of l1 after the first element with l2 and returns the replaced value.

A.3.2 General Operations

This section describes those procedures that allow the command interpreter to function as a generalized programming language. These include mathematics, logical, file, and string operations and routines for manipulating the compound data types. They also include routines for determining the type of a data object and for converting between types. Their specifications are as follows:

+ = **procedure** (numeric n1, numeric n2, ...) **returns** numeric
effects: + returns the sum of all arguments, or zero if the argument list is empty.

- = **procedure** (numeric n1, numeric n2, ...) **returns** numeric
effects: - returns the result of subtracting all arguments but the first from the first. Unlike +, - must have at least one argument.

* = **procedure** (numeric n1, numeric n2, ...) **returns** numeric
effects: * returns the product of all arguments, or 1.0 if the argument list is empty.

/ = **procedure** (numeric n1, numeric n2, ...) **returns** numeric
  effects: / returns the result of dividing the first argument by all remaining arguments. Unlike *, / must have at least one argument.

= = **procedure** (any a1, any a2) **returns** boolean
  effects: = returns true if a1 and a2 have the same type and are equal, false if they have the same type and are not equal, and an error if they do not have the same type. For compound objects, equal means that they point to the same location in memory. For simple objects, it means that they have the same value.

> = **procedure** (any a1, any a2) **returns** boolean
  effects: If a1 and a2 are of different types, > returns an error. If a1 and a2 are both numeric, it returns true if a1>a2, false otherwise. If a1 and a2 are both strings or both labels, > returns true if a1 is lexicographically greater than a2, false otherwise. For other types, the behavior is undefined.

< = **procedure** (any a1, any a2) **returns** boolean
  effects: If a1 and a2 are of different types, < returns an error. If a1 and a2 are both numeric, it returns true if a1<a2, false otherwise. If a1 and a2 are both strings or both labels, < returns true if a1 is lexicographically less than a2, false otherwise. For other types, the behavior is undefined.

acos = **procedure** (numeric n) **returns** numeric
  effects: acos returns the Arccosine of n radians.

and = **procedure** (boolean b1, boolean b2) **returns** boolean
  effects: and returns the logical and of b1 and b2.

asin = **procedure** (numeric n) **returns** numeric
  effects: asin returns the Arcsin of n radians.

atan = **procedure** (numeric n) **returns** numeric
  effects: atan returns the Arctangent of n radians.

atan2 = **procedure** (numeric n1, numeric n2) **returns** numeric
  effects: atan2 returns the Arctangent of n2/n1 radians.

bit2double = **procedure** (bitmap b) **returns** numeric
  effects: bit2double interprets b as a floating point number and returns the number.

bit2int = **procedure** (bitmap b) **returns** numeric
  effects: bit2int interprets b as an integer and returns its value.

closef = **procedure** (fileid f) **returns** fileid
  effects: closef closes f and returns it.

concat = **procedure** (list l1, list l2) **returns** list
  modifies: l1
  effects: concat concatenates l2 onto the end of l1.

cos = **procedure** (numeric n) **returns** numeric
  effects: cos returns the cosine of n radians.

debug = **procedure** (string s) **returns** string
  effects: debug outputs s to the standard output and returns s.

double2bit = **procedure** (numeric n) **returns** bitmap
  effects: double2bit returns a bitmap containing the floating point representation of n.

exp = **procedure** (numeric n) **returns** numeric
  effects: exp returns e^n.

int2bit = **procedure** (numeric n) **returns** bitmap
  effects: int2bit returns a bitmap containing the binary value of the integer part of n.

isarray = **procedure** (any a) **returns** boolean
  effects: isarray returns true if a is an array, false otherwise.

isboolean = **procedure** (any a) **returns** boolean
  effects: isboolean returns true if a is a boolean, false otherwise.

isfileid = **procedure** (any a) **returns** boolean
  effects: isfileid returns true if a is a fileid, false otherwise.

islabel = **procedure** (any a) **returns** boolean
  effects: islabel returns true if a is a label, false otherwise.

islist = **procedure** (any a) **returns** boolean
  effects: islist returns true if a is a list, false otherwise.

isnumeric = **procedure** (any a) **returns** boolean
  effects: isnumeric returns true if a is numeric, false otherwise.

isprocedure = **procedure** (any a) **returns** boolean
  effects: isprocedure returns true if a is a procedure, false otherwise.

isstring = **procedure** (any a) **returns** boolean
  effects: isstring returns true if a is a string, false otherwise.

lab2str = **procedure** (label l) **returns** string
  effects: lab2str returns the string associated with l.

length = **procedure** (list l) **returns** numeric
  effects: length returns the length of list l.

log = **procedure** (numeric n) **returns** numeric
  effects: log returns the natural log of n.

not = **procedure** (boolean b) **returns** boolean
  effects: not returns the logical negation of b.

num2str = **procedure** (numeric n) **returns** string
  effects: num2str returns a user-readable string form for n.

open = **procedure** (string f, string t) **returns** fileid
  effects: open opens a file of name f with type t, where t is the file type string as defined for the underlying C, and returns the file's id. If open cannot open the file, it returns an error.

or = **procedure** (boolean b1, boolean b2) **returns** boolean
  effects: or returns the logical or of b1 and b2.

pow = **procedure** (numeric a, numeric b) **returns** numeric
  effects: pow returns a^b.

readf = **procedure** (fileid f, string s) **returns** string
effects: readf reads in characters from f until it reaches the end of the file or finds a character in s. It returns a string containing all characters read including the termination character, if any.

sin = **procedure** (numeric n) **returns** numeric
effects: sin returns the sine of n radians.

size = **procedure** (array a) **returns** numeric
effects: size returns the number of elements a can contain.

str2lab = **procedure** (string s) **returns** label
effects: str2lab returns a label whose associated string is s.

str2num = **procedure** (string s) **returns** numeric
effects: If s contains the string representation of a number, str2num returns the number. Otherwise it returns an error.

strcat = **procedure** (string s1, string s2, ...) **returns** string
effects: strcat returns the result of concatenating all of its string arguments. It must have at least one argument.

strlen = **procedure** (string s) **returns** numeric
effects: strlen returns the number of characters in s.

substr = **procedure** (string s, numeric a, numeric b) **returns** string
effects: If $0 \leq a \leq b < |s|$, substr returns the substring of s with indices a to b, starting from 0, relative to s. Otherwise, it returns an error.

tan = **procedure** (numeric n) **returns** numeric
effects: tan returns the tangent of n radians.

trunc = **procedure** (numeric n) **returns** numeric
effects: trunc returns the integer portion of n.

writef = **procedure** (fileid f, string s) **returns** string
effects: writef writes s to f, provided f is open for writing, and returns s. If f is not open for writing, it returns an error.

A.3.3 Graphics Routines

This section describes routines for controlling the graphics window. These routines are used to draw some simple objects and control how they appear in the window. The reason some of these routines have the same name as hooks described in Section A.2 is that default functions are provided for these hooks. These default routines can, however, be replaced by user-defined routines in order to customize simulator behavior. The following are specifications for the graphics routines:

init_graphics = **procedure** (numeric wx, numeric wy, numeric wz, numeric ex, numeric ey, numeric ez, numeric cx, numeric cy, numeric cz, numeric ux, numeric uy, numeric uz) **returns** boolean

effects: init_graphics initializes the graphics display with workspace size (wx,wy,wz), eye point (ex,ey,ez), focal point (cx,cy,cz), and up vector (ux,uy,uz), then returns true.

make_cylinder = **procedure** (numeric sx, numeric sy, numeric sz, numeric ex, numeric ey, numeric ez, numeric r, numeric c) **returns** boolean

effects: make_cylinder draws a cylinder with axis from (sx,sy,sz) to (ex,ey,ez), radius r, and color c and returns true.

make_line = **procedure** (numeric sx, numeric sy, numeric sz, numeric ex, numeric ey, numeric ez, numeric c) **returns** boolean

effects: make_line draws a line from (sx,sy,sz) to (ex,ey,ez) with color c and returns true.

make_sphere = **procedure** (numerics cx, numerics cy, numerics cz, numerics r, numerics c) **returns** boolean

effects: make_sphere draws a sphere at center (cx,cy,cz) with radius r and color c and returns true.

redraw = **procedure** (numeric c) **returns** boolean

effects: redraw redraws the screen, setting the graphics to color c, and returns true.

rotate_x = **procedure** (numeric a) **returns** boolean

effects: rotate_x rotates the graphics counter-clockwise by a degrees around the X-axis and returns true.

rotate_y = **procedure** (numeric a) **returns** boolean

effects: rotate_y rotates the graphics counter-clockwise by a degrees around the Y-axis and returns true.

rotate_z = **procedure** (numeric a) **returns** boolean

effects: rotate_z rotates the graphics counter-clockwise by a degrees around the Z-axis and returns true.

zoom_in = **procedure** () **returns** boolean

effects: zoom_in moves the eye point to multiply the distance from the focal point to the eye point by 0.8 and returns true.

zoom_out = **procedure** () **returns** boolean

effects: zoom_out moves the eye point to multiply the distance from the focal point to the eye point by 1.25 and returns true.

A.3.4 Communication Routines

This section covers routines used by the command interpreter to communicate with the numerical simulator. Some of these routines are used to create new nodes and edges or new types for nodes and edges. Others instruct the numerical simulator to modify specific parameters. They also include routines to query the numerical simulator about its data. The specifications are as follows:

clear_all = **procedure** () **returns** boolean

effects: clear_all instructs the numerical simulator to clear all data and return to a startup state. It returns true.

connect = **procedure** (string h, numeric p) **returns** boolean
effects: connect attempts to establish a connection to the host h, checking at ports from p to p+31. It returns true if it establishes the connection, and an error if cannot.

disconnect = **procedure** () **returns** boolean
effects: disconnect closes the connection, if any, and returns true.

init_workspace = **procedure** (numeric x, numeric y, numeric z) **returns** boolean
effects: init_workspace instructs the numerical simulator to set the workspace size to (x,y,z) and returns true.

make_edge = **procedure** (numeric n1, numeric n2) **returns** boolean
effects: make_edge instructs the numerical simulator to attach edge n1 to edge n2 and returns true.

make_edge_type = **procedure** (list o, numeric ks, numeric kt, numeric kr, numeric er, numeric ed, numeric r, numeric a) **returns** boolean
effects: make_edge_type instructs the numerical simulator to define a new edge type. o is a list of numeric values specifying the other types to which this edge can attach. ks, kt, and kr are the three spring constants. er and ed are binding energies. r and a are the distance and angle tolerances, respectively. make_edge_type returns true.

make_node = **procedure** (numeric n, list l1, list l2, list l3, list l4) returns boolean
effects: make_node instructs the numerical simulator to create a node of type n with position, velocity, angle, and angular velocity given by l1, l2, l3, and l4, respectively, and returns true.

make_node_type = **procedure** (list l) **returns** boolean
effects: make_node_type instructs the numerical simulator to define a new type of node. The first element of the list is one of "single", "complex", and "variable", for defining single, complex, and variable nodes, respectively. For single nodes, the remainder of the list specifies the radius, a currently unused numerical parameter, the mass, and lists of the positions, up vectors, and types of the edges. For complex nodes, the remainder of the list consists of lists of the offsets, rotations, and types of the sub-nodes. For variable node types, the remainder of the list is two lists, the first listing conformation energies, and the second listing the corresponding conformations. make_node_type returns true.

query_all = **procedure** () **returns** list
effects: query_all requests simulator data on the positions of all nodes and returns it in the form of a list whose first element is an array of the indices of all single nodes, whose second element is an array of their positions, whose third element is an array of their orientations, and whose final element is an array of arrays of the status of their edges, in which an element is -1 if the corresponding edge is unconnected and the index of the edge to which it is connected, otherwise.

query_confs = **procedure** () **returns** array
effects: query_confs queries the numerical simulator about the conformation of each variable node and returns an array of all conformations.

query_constant = **procedure** (string s) **returns** numeric
effects: query_constant queries the numerical simulator about the value named by s and returns the result.

query_edges = **procedure** () **returns** array
effects: query_edges queries the numerical simulator about which edges are connected and returns an array in which each element i is -1 if edge i is not connected to any other edge and is j if edge i is connected to edge j.

query_node = **procedure** (numeric n) **returns** array

effects: query_node queries the numerical simulator about the position, velocity, orientation, and angular velocity of node n and returns an array of twelve numeric values containing that data, in order.

remove_edge = **procedure** (numeric n) **returns** boolean
effects: remove_edge instructs the numerical simulator to break any bond containing edge n and returns true.

remove_node = **procedure** (numeric n) **returns** boolean
effects: remove_node instructs the numerical simulator to remove the node of index n and returns true.

set_conformation = **procedure** (numeric n1, numeric n2) **returns** boolean
effects: set_conformation instructs the numerical simulator to set variable node number n1 to conformation n2 and returns true.

set_constant = **procedure** (string s, numeric n) **returns** boolean
effects: set_constant instructs the numerical simulator to set the value named by s to n and returns true.

step = **procedure** () **returns** boolean
effects: step instructs the numerical simulator to advance the time by steps_per_update·timestep and returns true.

A.3.5 Vector Routines

Although vectors are not a primitive type of the control language, there are several primitive routines for dealing with them. Vectors are represented by arrays of three numeric elements. Therefore, any of the vector math operations could be implemented in terms of the primitives already presented. These primitive routines were created only to improve efficiency; vector operations are very common and are crucial to good performance of the command interpreter, so hard-coding them can improve performance substantially. The following are specifications of the primitive vector operations:

compose_rotations = **procedure** (vector r1, vector r2) **returns** vector
effects: compose_rotations returns a rotation vector whose effect is equivalent to rotating by r1 followed by rotating by r2. Rotating by a vector is defined as rotating counter-clockwise around the vector by a number of radians equal to the length of the vector.

svprod = **procedure** (numeric n, vector v) **returns** vector
effects: svprod returns the vector produced by multiplying v term-wise by n.

vadd = **procedure** (vector v1, vector v2) **returns** vector
effects: vadd returns the vector sum of v1 and v2.

vcross = **procedure** (vector v1, vector v2) **returns** vector
effects: vcross returns the cross product of v1 and v2.

vdot = **procedure** (vector v1, vector v2) **returns** numeric

effects: vdot returns the dot product of v1 and v2.

vecrot = **procedure** (vector v, vector r) **returns** vector
effects: vecrot returns the result of rotating v around r counter-clockwise by $||r||$ radians.

vecx = **procedure** (vector v) **returns** numeric
effects: vecx returns the X component of v.

vecy = **procedure** (vector v) **returns** numeric
effects: vecy returns the Y component of v.

vecz = **procedure** (vector v) **returns** numeric
effects: vecz returns the Z component of v.

vlen = **procedure** (vector v) **returns** numeric
effects: vlen returns the length of v.

vnorm = **procedure** (vector v) **returns** vector
effects: vnorm returns a vector with the same direction as v and a length of one.

vrotx = **procedure** (vector v, numeric a) **returns** vector
effects: vrotx returns the result of rotating v counter-clockwise by a degrees around the X axis.

vroty = **procedure** (vector v, numeric a) **returns** vector
effects: vroty returns the result of rotating v counter-clockwise by a degrees around the Y axis.

vrotz = **procedure** (vector v, numeric a) **returns** vector
effects: vrotz returns the result of rotating v counter-clockwise by a degrees around the Z axis.

vsub = **procedure** (vector v1, vector v2) **returns** vector
effects: vsub returns the vector difference of v1 and v2.