Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1994-09

Modeling of real-time dynamic effects

Watt, Anne E.

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/43039

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A286 017

DTIC
ELECTE
NOV 0 8 1994
S D
F

THESIS

MODELING OF REAL-TIME DYNAMIC EFFECTS

by

Anne E. Watt

September 1994

Thesis Advisor: David R. Pratt
Co-Advisor: Michael J. Zyda

Approved for public release; distribution is unlimited.

1180

94-34562

94 11 7 026

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1994	Master's Thesis

4. TITLE AND SUBTITLE
MODELING OF REAL-TIME DYNAMIC EFFECTS (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Watt, Anne E.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Distributed three dimensional combat simulation systems such as the Naval Postgraduate School's NPSNET project lack many of the characteristic effects of the live battlefield. This deficiency is the problem we sought to eliminate. Our approach to solving this absence of effects was to evaluate previous work performed in this area and incorporate aspects of this research that would assist in creating believable effects capable of running in real-time.

This thesis focuses on simulating three elements of these war zones - realistic clouds both from an internal and external viewpoint which move due to gridded wind vectors, incorporation of a recording and transmission process for dynamic terrain effects through the implementation of the Distributed Interactive Simulation (DIS) protocol's recently approved Destructible Entity protocol data units (PDUs), and physically-based explosions. The result of this research is a set of effects' simulators available for further studying of and experimenting with modifications to these implementations. These programs also provide users with frame rate feedback regarding their modifications to the effects. Furthermore, the cloud implementations and explosive effects are too computationally expensive to be incorporated within complex simulators such as NPSNET.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Graphics, Clouds, Distributed Interactive Simulation (DIS) protocol, Dynamic Terrain, Explosions	124
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Approved for public release; distribution is unlimited

MODELING OF REAL-TIME DYNAMIC EFFECTS

Anne E. Watt
Lieutenant, United States Navy
B.S., United States Naval Academy, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1994

Accesion For
NTIS CRA&I ☒
DTIC TAB ☐
Unannounced ☐
Justification .....
By .....
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

Author: [redacted]
Anne E. Watt

Approved By: [redacted]
David R. Pratt, Thesis Advisor

[redacted]
Michael J. Zyda, Thesis Co-Advisor

[redacted]
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Distributed three dimensional combat simulation systems such as the Naval Postgraduate School's NPSNET project lack many of the characteristic effects of the live battlefield. This deficiency is the problem we sought to eliminate. Our approach to solving this absence of effects was to evaluate previous work performed in this area and incorporate aspects of this research that would assist in creating believable effects capable of running in real-time.

This thesis focuses on simulating three elements of these war zones - realistic clouds both from an internal and external viewpoint which move due to gridded wind vectors, incorporation of a recording and transmission process for dynamic terrain effects through the implementation of the Distributed Interactive Simulation (DIS) protocol's recently approved Destructible Entity protocol data units (PDUs), and physically-based explosions. The result of this research is a set of effects' simulators available for further studying of and experimenting with modifications to these implementations. These programs also provide users with frame rate feedback regarding their modifications to the effects. Furthermore, the cloud implementations and explosive effects are too computationally expensive to be incorporated within complex simulators such as NPSNET.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

B. OBJECTIVES 2

C. ORGANIZATION 3

II. PERFORMER, DIS, AND NPSNET 5

A. PERFORMER 5

B. DIS 6

C. NPSNET 7

D. SUMMARY 8

III. EFFECTS RESEARCH 9

A. OBSCURANTS AS THEY RELATE TO CLOUDS 9

1. Corbin 9

2. Phillips Laboratory, TASC, and LORAL 10

3. Obscurants Summary 13

B. DYNAMIC TERRAIN 13

1. Walters 13

2. Virtual Bulldozer 15

3. Dynamic Terrain Architectures 16

4. Dynamic Terrain Summary 20

C. EXPLOSIONS 20

1. Nash 20

2. Monahan 21

3. Explosions Summary 23

D. SUMMARY OF EFFECTS RESEARCH 23

IV. CLOUDS 25

A. CLOUD CHARACTERISTICS 25

B. CLOUD MODELER 27
1. Puff Characteristics and Construction 27
2. Lighting 29
3. Resulting pfGeoState Associated With the Cloud Puff 30
4. Adding the Puff to the Scene Graph Hierarchy 30
5. Cloud Modeler's Capabilities 33
6. Cloud Puff Distributions 35
7. Performance of and Applications for the Cloud Modeler 39
C. MODELING THE STRATUS LAYER 41
1. Characteristics 41
2. Stratus Layer in the Cloud Modeler 42
3. Stratus Layer in NPSNET 43
D. CLOUDS AFFECTED BY WINDS 44
E. CLOUD SUMMARY 49
V. NETWORKING DYNAMIC TERRAIN 51
A. DIS VERSION 3.0 DESTRUCTIBLE ENTITY PDUs 52
1. Create/Modify PDU 53
2. Deletion PDU 54
3. Request ID PDU 56
4. Reply ID PDU 57
5. Request Object PDU 58
6. Reply Object PDU 59
B. DESTRUCTIBLE ENTITY PDU IMPLEMENTATION 61
1. Destructible Entity Functions File 63
2. Player and Terrain Manager Interaction 71
3. Destructible Entity PDU Performance 75
C. DESTRUCTIBLE ENTITY PDU SUMMARY 76
VI. EXPLOSIONS 77

A. INTRODUCTION TO THE EXPLOSION FUNCTION 77
1. Function's Parameters 78
2. Performer pfGeoSet Functions 79
B. EXPLOSION FUNCTION DETAILS 80
1. Explodable Format Conversion Algorithm 81
2. Explosion Motion Equations 87
C. PERFORMANCE WITHIN A TEST APPLICATION 91
D. EXPLOSIONS SUMMARY 93
VII. CONCLUSION AND TOPICS FOR FURTHER RESEARCH 95
A. CONCLUSION 95
B. FUTURE RESEARCH 95
APPENDIX. USER'S GUIDES 97
A. CLOUD MODELER 97
1. How to Start the Program 97
2. Input Devices for Designing and Viewing a Cloud 97
B. WIND VECTOR SIMULATOR 102
1. How to Start the Program 102
2. Input Devices for Viewing the Clouds Within the Wind Vector World. 102
C. DIS DESTRUCTIBLE ENTITY PDU IMPLEMENTATION PROGRAM. 103
D. EXPLOSIONS TEST HARNESS 104
1. How to Start the Program 104
2. Key Bindings 104
LIST OF REFERENCES 107
INITIAL DISTRIBUTION LIST 109

LIST OF FIGURES

1. 8-sided Polygon Built Using Tristrips 28
2. Creating a pfGeoState to Attach to pfGeoSet 31
3. Cloud Modeler Scene Hierarchy 32
4. Uniform Distribution Drawing Order 36
5. Uniformly Distributed Cloud - 100 Puffs, 4 Rows, and Depth of 3 37
6. A Standard Normal Distribution Curve 38
7. Cloud With Normal Distribution of 100 Puffs and Depth of 10 38
8. The Symmetrical Normal Distribution 39
9. Cloud With Symmetrical Normal Distribution of 100 Puffs and Depth of 10 40
10. Performer's Layered Atmospheric Model 43
11. Winds Within the Wind Vector Simulation 45
12. The Scene Hierarchy for the Wind Vector Simulator 46
13. Default Cloud 47
14. Structures that Support the Use of Destructible Entity PDUs 64
15. Functions that Support the Use of Destructible Entity PDUs 65
16. Example Host Matrix 66
17. Destructible Entity Request and Reply PDU Sequence 72
18. Creation or Modification of a Destructible Entity 73
19. Example of Obtaining and Modifying pfGeoSet Data 82
20. Algorithm Used to Convert Tristrips to Independent Triangles 84
21. Computing the Normal of a Triangle 85
22. Computation of Instantaneous Velocity for Triangle Primitives 89
23. Ballistic Motion Equations 90
24. Helicopter Before Impact 92
25. Helicopter After Impact 93

LIST OF TABLES

1. Cloud Resolution Vs. Polygon Count And Frame Rate From Ref. [LADS94] 12
2. Frame Rate And Viewpoint Altitude Table From Ref. [LADS94] 13
3. Cloud Types and Associated Conditions After Ref. [NAVA79] 26
4. Applicable Iris Performer Node Types After Ref. [SGIA94] 31
5. Performance Tests of Clouds Composed of Uniformly Distributed Puffs 40
6. Create/Modify PDU 55
7. Deletion PDU 56
8. Request ID PDU 57
9. Reply ID PDU 58
10. Request Object PDU 60
11. Reply Object PDU 62

I. INTRODUCTION

A. BACKGROUND

With more and more military training simulators becoming available and many military leaders looking toward these systems as a solution to the constant budget cuts the armed forces are being required to handle, accurate realistic modeling within synthetic environments is becoming absolutely imperative. Various virtual battlefield simulators incorporate realistically modeled vehicles and weapons, physically-based vehicle and weapon motion, vast terrain databases, and even autonomous agents; however, most lack the incorporation of more subtle effects that truly make a difference in actual war time situations. These effects include the simulation of meteorological occurrences such as the presence of clouds, wind, and precipitation, dynamic terrain features, and explosive effects. For example, the amount of clouds and their density present in the environment play a major role in what the warfighter is able to see approaching in the distance or in what sensors are able to perceive. In addition, adverse weather conditions can slow a unit down significantly and force unit commanders to have to devise alternative plans, sometimes within a moments notice. Therefore, if designers of synthetic environments neglect to model such occurrences, then trainees depending heavily on combat simulators to maintain their high state of operational readiness will find themselves under prepared to handle such situations during actual war time.

Beyond the concept of just modeling these effects, is the issue of networking them. Having the ability to conduct distributed simulations across long distances is one of the main factors that brought about this great deal of enthusiasm for using synthetic environments as an effective means of training troops; for they save a great deal of money, time, and complicated planning that normally are invested in bringing widespread units together for training exercises. Therefore, because networking is a key component to the success of such training systems, these effects must also be communicated to all hosts in

the simulation so that all participants are dealing with the same environmental circumstances and no one has an unfair advantage that one would not have in wartime.

Perhaps as important as networking, is the concept of immersion. According to [ZYDA93], in order to have a totally immersive system the graphics representing the virtual world must respond to the actions and movements of the user via input devices, and the virtual world must convince the user that he or she is really in the environment represented by the system. Accurate modeling of effects supports both of these requirements. For example, when an operator uses an input device whether it be the keyboard or a button on a flight control stick to shoot another entity, he or she expects to see as the result of a hit, the inflicted object exploding into many fragments and/or flames. If the user is not provided such a visual cue and others during the course of the simulation, then the system is not doing its job of convincing the user that he or she is actually in the virtual world. It is this feeling of immersion that enables a user to benefit the most from his or her training experiences in synthetic environments.

B. OBJECTIVES

The objective of this research was to develop dynamic effects for real-time distributed virtual world simulators. This research included modeling as well as networking such occurrences. Specifically, we sought to model realistic clouds (including their movement) and physically-based explosions, and to network dynamic terrain features using a set of protocol data units (PDUs) recently developed for the Distributed Interactive Simulation (DIS) protocol. Furthermore, our goal was to implement these effects in the Naval Postgraduate School's real-time distributed virtual world simulator, NPSNET. In addition, NPSNET incorporates Silicon Graphics Inc.'s software enhancing application program interface (API), Performer, and therefore our work was designed using Performer. NPSNET, Performer, and DIS are all further described in Chapter II.

C. ORGANIZATION

The following outlines the topics discussed in the remaining chapters of this document. Chapter II provides an overview of NPSNET, Performer, and DIS. Chapter III discusses previous work completed related to our objectives. Chapters IV, V, and VI describe our implementation of clouds, networked dynamic terrain, and explosions respectively. Chapter VII summarizes our work and provides topics of further research related to dynamic effects. Finally, the Appendix provides user's guides for the programs we developed to test our models.

II. PERFORMER, DIS, AND NPSNET

This chapter provides an overview description of Silicon Graphics, Inc. programming toolkit Performer, the Distributed Interactive Simulation (DIS) network protocol, and the Naval Postgraduate School's distributed virtual battlefield simulator, NPSNET, all of which play vital roles in the work presented in this thesis. Further implementation details regarding these areas are provided as they pertain to the research described in each of the following Chapters.

A. PERFORMER

Performer is a software toolkit that is used for creating and/or enhancing visual simulation applications [SGIA94]. In addition, applications are often built using Performer in order to take advantage of its ability to optimize the simulation's performance on Silicon Graphics systems. This application development package consists of two libraries, libpf and libpr. Libpf is a visual simulation library which provides such features as multipipe and multichannel capability, multiprocessing, frame control, and hierarchical scene graph construction which supports Performer's run-time database structure. Libpr is a low-level library which furnishes optimized rendering functions, state control, high-speed math routines and a variety of other functions necessary for implementing real-time graphics.

A key function of Performer is its multiprocessing capability which often times is implemented and managed solely by this toolkit unbeknownst to the programmer. Libpf provides a pipelined multiprocessing model where the pipeline stages consist of simulation, culled traversal, and draw. The simulation stage updates and queries the scene while the culled traversal process scans the database hierarchy for objects that are potentially visible and adds those entities to the display list. Finally, the draw process renders the geometry within the display list.

Another distinguishing feature of Performer is its run-time database structure - it does not define an archival database or file format. This database structure is built using a hierarchial graph called a directed acyclic graph. The hierarchy describes the relationship

of various objects through the use of inheritance among its nodes and in its spatial and logical organization. Logical organization means like items are grouped together, but this type of structuring does not always provide the best performance. Spatial organization on the other hand, allows for optimal culling and intersection testing since nodes are organized by areas of the world. The nodes available for the construction of this graph each have a specific function which coupled with the inheritance property provide potential for a wide variety of object definitions and relationships. Furthermore, although the graphs are typically built during load time, these nodes can be queried and updated at any time during the simulation.

Finally, Performer also allows a designer to construct objects using Performer's low level geometry. These geometric definitions use tuned rendering loops to eliminate the CPU bottleneck that often limit many Graphics Library applications. However, a programmer does not have to design within the Performer environment to benefit from this enhanced performance, rather this toolkit provides several routines which import popular database formats into Performer run-time structures.

Performer provides a wide variety of other desirable application development features; however, there exist far too many to be explained in this document. The above described features highlight the common reasons why this toolkit is chosen for enhancing or developing graphics applications. Further information regarding this toolkit is located in [SGIA94] and [SGIB94].

B. DIS

DIS is a networking protocol which has been and still is being developed to support the multitude of military simulation and training systems that are continually becoming available. It was designed with specific goals and missions in mind which are presented below and are the guidelines we used in our implementation of Destructible Entity PDUs.

"The primary mission of DIS is to create synthetic, virtual representations of warfare environments by systematically connecting separate subcomponents of simulation

which reside at distributed, multiple locations" [ISTA93]. Simply stated, the goal of DIS is to provide interoperability among different simulated warfare environments. The DIS protocol mandates that these separate subcomponents or nodes, as they are also called, control and maintain the state of one or more entities. Thus, a node "owns" certain entities and it is responsible for communicating their state changes to the other nodes connected to the simulation. This concept of hosts communicating entity state information eliminates the need for a central computer for event scheduling meaning that the potential for a single point of failure does not exist. Another important concept of the DIS protocol is that all nodes are responsible for interpreting and responding to messages (protocol data units - PDUs) from other nodes and for maintaining the status of the simulation's world and all of its entities. It is therefore up to a node to determine if information is relevant to its part of the world and if so, to update its environment so that it is consistent with those of the other participants. Finally, dead reckoning of entities by hosts between receipt of entities' update information (last reported location, velocity, and orientation) is also included in this standard as a means of limiting the network traffic.

By incorporating these concepts, the DIS protocol has provided a robust highly structured communication system that has allowed many heterogenous applications to interact together in real-time. This protocol as previously mentioned is still in the development and testing phase, for there are many aspects of the wartime environment left to be handled in order to provide a communications protocol that supports an effective and believable distributed training system. One of these aspects is terrain modifications and thus, we sought to implement the newly emerging DIS Destructible Entity PDUs.

C. NPSNET

The NPSNET project is a three-dimensional real-time networked combat simulator developed at the Naval Postgraduate School's (NPS) Department of Computer Science [ZYDA93]. It is designed to run on low cost graphics workstations such as the Silicon Graphics IRIS family of computers and communicates with other nodes on the network

using Simulation Network (SIMNET) and DIS networking formats. In addition, NPSNET uses Performer in order to take advantage of the optimized system performance features it provides. This simulator allows users to interact with three-dimensional terrain, structures, and other players within the world. The goal of the project is to develop a fully interactive, believable environment and thus, this project incorporates a wide variety of diverse research areas. These topics of research include real-time scene management, physically-based modeling, dynamic terrain, autonomous forces and expert systems, meteorological and atmospheric effects, and DIS integration and development, to name just a few. Therefore, NPSNET also provides an excellent platform upon which various concepts related to virtual simulators may be experimented with and studied.

D. SUMMARY

Our research efforts centered around the three areas presented in this chapter. Our objective was to model effects for NPSNET to support the project's goal of providing a truly believable environment. Due to Performer's readily available multitude of application enhancing features and its incorporation in NPSNET all of our work was designed using Performer. Additionally, to further the integration of the DIS protocol within various military simulators and to enhance NPSNET's handling of dynamic terrain our final objective was to design a library of functions for implementing the tentatively approved DIS Destructible Entity PDUs.

III. EFFECTS RESEARCH

The research presented below describes related computer simulated clouds, dynamic terrain, and explosion implementations that have been inserted in three-dimensional (3D) simulators; however they all have certain limitations, many of which we sought to eliminate. In addition, several of the ideas presented in this chapter were incorporated in our efforts to solve these problems. Their precise use is explained in following implementation chapters.

A. OBSCURANTS AS THEY RELATE TO CLOUDS

1. Corbin

Daniel Corbin developed a library of functions called the Environmental Effects Library which includes atmospheric obscurants - smoke, vehicle dust trails, and clouds - that are implemented in real-time in NPSNET [CORB93]. Basically the same characteristics and modeling features are used for all three obscurants. These cloud variations are generated using plumes that consist of five columns of continuous puff primitives translated along a centerline. The puff primitive is made of three perpendicularly aligned 12-sided polygons each textured by a red, green, blue, and alpha (RGBA) image file. The concept of a particle system is incorporated meaning that the puff primitives maintain their own attributes of age, position, size, shape, and translucence, and are translated as a function of their attributes. Thus, any singular puff is considered to be insignificant, but, the puffs as a group define the resulting shape and appearance of the plume.

In addition, the age of the puff (elapsed time since its creation) is a very important concept in that it determines the position, size, and opacity of each of the puffs. The maximum obtainable puff age is a function of both the amount of puffs available for the current obscurant and the maximum downwind distance that the puff is expected to traverse. As the puff's age increases (moves away from generating source), it also appears

to fade away since the texture image file is blended with the RGBA characteristics of the underlying polygon whose alpha value decreases with age.

Plume generation takes place by continually assigning puffs to one of three states - the unused queue, the active queue, and the transformation and rendering phase. Initially all puffs are assigned to the unused queue waiting to become part of the obscurant column. When this event occurs, the puff's attributes are updated and it is translated the required amount of distance. It is then scaled and placed back on the active queue. A puff on the active queue is continually popped off, rendered in the obscurant column, updated, and pushed back on the active queue until it reaches its maximum designated age at which point it is returned to the unused queue. Therefore, the obscurant is visible for as long as there remain puffs that are still "alive."

This implementation provides for realistic obscurant models within a 3D simulator where precise physical accuracy is not imperative. However, motion is only simulated as it relates to smoke and vehicle dust trails; whereas clouds are initially placed in random positions in the world where they remain for the entire simulation and thus, are not affected by the ambient air mass surrounding them.

2. Phillips Laboratory, TASC, and LORAL

The U.S. Air Force's Phillips Laboratory and The Analytical Sciences Corporation (TASC) jointly developed a high-fidelity model for atmospheric clouds [LADS94]. The model generates a 3D volume grid of data that is converted into visual primitives that form the clouds which extend horizontally as far as 50 km and have a resolution as little as 10 m. These simulated clouds are very realistic, however require hundreds of thousands of polygons preventing the use of this modeler within a real-time synthetic environment.

The U.S. Army Topographic Engineering Center (TEC) in Ft. Belvoir, Virginia launched an effort to modify this model as little as possible so that extremely realistic clouds could be placed in a real-time DIS environment. This challenge was assigned to both TASC and Loral Advanced Distributed Simulation. "Simple 3D oriented 2D disk-like

stamps (oriented toward the viewpoint)" were developed to form a variety of clouds (cumuliform) that are capable of being rapidly rendered [LADS94]. Cloud formations are displayed by converting 3D grid data into primitives that are rendered at their appropriate locations. More specifically, a Cloud Generator (based on TASC's original Cloud Scene Simulation Model program and controlled by the simulation host) accepts cloud parameters from an environmental PDU, sounding data, and other pertinent information and produces 3D grid cloud data (based on liquid water content). It also incorporates a predictive model of cloud data for smooth interpolation between time periods. The Visual Data Generator and Volumetric Manager (controlled by the graphics processor) then convert this information into the appropriate primitive data so that all primitives may be rendered by the graphics hardware. Additionally, there is an entity on the network that acts as the environmental manager meaning that it is responsible for controlling the overall environment and for periodically issuing an environmental PDU. This environmental PDU is the one that has been proposed in the DIS workshop with one exception. In order for the Cloud Generator to be able to use the information in the environmental PDU, additional data fields had to be added to the PDU that are required to start the Cloud Scene Simulation Model program. This model specific addition, however, means that if the DIS environmental PDU remains as is, then this simulation would not be able to operate in conjunction with multiple distinct DIS platforms running together. Another issue of concern regarding DIS compatibility, is that the modeler requires that cloud data be calculated for the entire geographic database at initialization; whereas most DIS applications "require a virtual window of interest within the larger geographic database" [LADS94]. Finally, although the cloud model uses sounding data to incorporate wind associated with the clouds, it does not use wind vector gridded data.

The modeler runs in real-time within a 3D simulator such as NPSNET, but within certain restrictions related to fractional coverage, polygon count, resolution, and viewing perspective. The tables below illustrate the frame rates obtainable (real-time performance occurs at a minimum of 10 frames per second) at varying degrees of these limitations. Table

1 consists of two cloud tests' results taken from a viewing range of 4 km under most of the clouds and shows that level-of-detail modeling based on proximity of the viewpoint is a way to substantially improve the frame rate of the simulation. Table 2 displays data from two additional tests where the user was moving through the cloud layers. Tests 3 and 4 involved a viewpoint moving normal and parallel to the cloud layer respectively. These tests illustrate that simulating flight through clouds is an area still in need of further research and experimentation in order to achieve consistent real-time response especially if a user were to fly parallel through dense fog areas like those that exist on the west coast of the United States.

Tests 1 and 2 Conditions:

- 10 km by 10 km cloud extents, 1000 m base and 2000 m top
- Cumulus clouds

Test Number	Fractional Coverage	X-Y Cloud Resolution	Cloud Polygons	Frame Rate
1	0.25	1000	84	30
	0.25	500	509	8.6
	0.25	100	8358	1
2				
	0.75	1000	254	15
	0.75	500	1174	3
	0.75	100	22674	0

Table 1: Cloud Resolution Vs. Polygon Count And Frame Rate From Ref. [LADS94]

Test 3 Conditions:

- 10 km by 10 km cloud extents, 1000 m base and 2000 m top
- 0.25 fractional coverage, cumulus, 500 m, XY cloud resolution

Test 4 Conditions:

- 10 km by 10 km clouds extents, 1000 m base and 2000 m top
- 0.75 fractional coverage, cumulus, 1000 m XY cloud resolution

Viewpoint Altitude (meters)	Frame Rate Test 5	Frame Rate Test 6
0	10	7.5
500	8.6	6.7
1000	10	6.7
1500	12	8.6
2000	15	8.6
8000	20	10

Table 2: Frame Rate And Viewpoint Altitude Table From Ref. [LADS94]

3. Obscurants Summary

As a result of studying the above described obscurant simulations we felt that in order to achieve our primary goal of real-time physically-based cloud simulation in NPSNET we needed to design a cloud modeler. We constructed the modeler with the particle system in mind and incorporated Corbin's primitive so as to allow the user to change its physical properties (ambient, diffuse, and alpha components) and to designate a desired distribution of multiple puffs. In addition, we felt it necessary to create a world of simulated wind vectors whose magnitude and direction vary with location in the world. This simulator serves as a testbed for newly modeled clouds created using the modeler.

B. DYNAMIC TERRAIN

1. Walters

Alan Walters implemented dynamic terrain features - craters, berms, and bridges - in NPSNET [WALT92]. Additionally, his work involved ensuring vehicles realistically respond to these earthworks by rolling and tilting when encountering uneven terrain or falling off a bridge when driving over the edge. Each earthwork is a C++ class derived from

a common base class. The terrain (made of grid squares with elevation post corners) has information holders associated with each of its squares that maintain a linked list of reference pointers to all earthworks contained or partially contained within the corresponding square. The earthwork, is responsible for handling its own details - creation, display, and interaction with vehicles. This approach is especially important in terms of eliminating common problems that often times occur when the terrain controls the earthworks (especially those that extend over many squares) such as the simulator's inability to display partially visible earthworks due to the fact that the part of terrain controlling them is not visible during terrain culling. Instead, each visible grid square tells all earthworks pointed to in their linked list of reference pointers to display themselves.

Another important aspect of Walters' work is that the earthworks are placed on top of the ground rather than as modifications to the underlying terrain. Dynamic terrain features such as craters and trenches definitely appear more realistic if the terrain is changed; however, such terrain modification methods as adding micro grid squares to each standard square or completely changing the grid squares and then having to render after every creation or modification to the terrain proved to be far too computationally intensive for real-time simulations. By placing the earthworks on top of the terrain and using the linked list of reference counting pointers, the terrain database does not have to be modified (and continuously rendered over and over again), and thus, the only changes that need to be made are additions to and deletions from the earthworks linked lists.

A limitation of this implementation is that although dynamic terrain features are modeled in real-time within NPSNET, they are not currently networked meaning that when other DIS simulators run in conjunction with NPSNET, they do not have identical displays for the same area of the common terrain database - an extremely important criteria for virtual battlefield simulators. However, a set of PDUs for the DIS protocol have been recently approved (DIS protocol version 3.0) called destructible entity PDUs and it is the implementation of these PDUs within NPSNET that we sought to incorporate.

2. Virtual Bulldozer

Xin Li and Michael Moshell from The Institute for Simulation and Training (IST) focused their attention on excavating activities - digging, cutting, piling, carrying, and dumping soil [LI93]. Initially, they developed a virtual bulldozer for moving terrain in front of a blade thereby creating a berm. The berm appears physically realistic since it is smoothed by a bidirectional Cardinal spline algorithm; however, it is strictly a kinematic model with no volume conservation implemented. Also, there are no forces computed and the soil does not slump when the bulldozer leaves. Therefore, an effort was launched by Jennifer Burg and Moshell [BURG91] to simulate soil piling up and consequently spilling down. This model is still kinematic, but is volume conserving during certain conditions. In addition, it does not consider the physical properties of the different types of soils.

Li and Moshell next focused their efforts on producing dynamic models of soil slippage and soil manipulations. After researching soil properties, it was determined that it would be too computationally expensive to model dynamics of various soil types due to the many environmental conditions that significantly influence them - moisture content, pore pressures, structural disturbances, fluctuation in the ground water table, underground water movements, stress history, time, and chemical action. They felt speed and realistic appearance were more important than simulating constantly changing internal soil stresses and therefore, the soil used in this research is considered to be homogeneous and isotropic. Their efforts resulted in the simulation of two excavating machines, a bulldozer and scooploader. The interaction between the soil and this equipment (not only pushing, but also scooping soil) is physically modeled by determining if the soil configuration is in static equilibrium, calculating the forces which drive a portion of the soil to slide if the configuration is not stable, and at the same time preserving the volume conservation. These simulations do indeed run in real-time. On a Silicon Graphics 4D/240 GTX computer, two bulldozers operate using four processors at 6-8 frames per second (not quite real-time - 10 frames per second is considered the minimal real-time rate) and the scooploader runs at 10-15 frames per second using two processors. Thus, these extremely computationally

intensive models at best barely run in real-time on their own. Furthermore, if they were implemented in conjunction with a 3D simulator like NPSNET, the simulator would in no way achieve real-time performance. The reason for this decline in performance is that these simulators already struggle to handle vast amounts of computations and network communications such that if coupled with these models would just make the system's load overwhelming.

3. Dynamic Terrain Architectures

IST has taken a look at what factors are crucial in being able to simulate dynamic terrain in real-time. They feel that a great deal of the limitations are due to the current data representation methods that require entirely too much storage and therefore, results in the transmission of a vast amount of data over a networked system like DIS just to describe terrain changes [IST94]. Additionally, IST feels that there needs to be a software architecture that supports the many mathematical models with varying data representations that have been developed to represent dynamic terrain.

As a result, IST launched an effort to determine what issues must be considered when designing such a flexible architecture and in doing so they progressively designed and evaluated their own architectures, each one an improvement over the previous one [LISL94]. Their first consideration involved the need for a central terrain or environmental manager versus a fully-distributed system. Although the DIS protocol insists that there be no central server, IST came to the conclusion that perhaps it is better to consider the applications on a case by case basis and let the applications' needs determine which method of terrain management must be employed.

Data structures was the next issue to be evaluated. Although most simulation and image generation systems use a polygonal representation for their terrain, it is felt that this modeling method is fine for rendering (viewing) in high-performance graphics systems, but, is an inadequate method for terrain following algorithms. The inadequacy lies in the fact that polygonal representations do not provide smoothly changing terrain slope

contours; rather slope changes are drastic at polygonal boundaries. These sudden changes cause vehicles driving over the terrain to react in a way contrary to what the viewer expects. Therefore, mathematical surfaces (B-splines) were employed as a viable alternative since they are capable of being sampled anywhere not just at data points, and are still compatible with polygonal systems.

The Dynamic Terrain Database (DTDB) was designed next in an effort to support a multitude of soil attributes (such as elevation, soil strength, temperature, and water depth) conceptually using multiple planes or layers for a particular area in the terrain database. These "multi-planes" are represented by a linked list (0..n attributes) of information. A client of the database does not know how the data is stored, but may query the database for information or change the value of one of the attributes at any point. In addition, queries of extensive areas may also be made which involve performing interpolations between the data points. Having the ability to make such queries is definitely a significant advantage of this database especially since such vast areas often times encompass a variety of different underlying representations. Another advantage of this database is that it serves as a local active database for each simulation node on a DIS network meaning that each node has the ability to maintain its own terrain database in a similar way that it keeps track of various active entities on the network (employing dead reckoning and occasional broadcasts of terrain state).

IST's design of system-level architectures began with one that was designed to obtain as much information related to the problems with incorporating dynamic terrain within DIS networked simulators. The designers were especially concerned with the areas of network bandwidth, CPU performance, and the possibility of maintaining different geographic areas on distinct CPUs. There are three important components within this architecture. The first component is the Dynamic Terrain Portal (DTP) which serves as the only connection to the DIS network and the internal dynamic terrain architecture. The DTP has three important responsibilities - it examines DIS traffic for dynamic terrain events, handles overall dynamic terrain simulation details (interaction between the other two

components), and sends new dynamic terrain messages to simulators. The second component encompasses the Dynamic Terrain Resources (DTRs). The DTRs each contain a specific physics-based algorithm for simulating the terrain features (such as soil slippage) and as new algorithms are developed they may be added to the overall architecture as DTRs. Finally, the last important part of the architecture is the database itself which has already been discussed above.

Overall, this initial architectural design provided important feedback. The DTP quickly became a bottleneck since it was the only interface between DIS and the dynamic terrain models. It was determined that the DTP, DTRs, and database all need to be run on the same CPU or on a shared memory multiprocessor due to the large amounts of data transfer between these three components. Furthermore, the developers realized the importance of the design of the database in terms of system performance especially as it relates to access conflicts to the single shared database.

The next architectural design is based on the previous one, but it also focuses more on the implementation of the terrain manager and making improvements in the areas of eliminating the bottleneck described above, incorporating more parallelism, and isolating jobs related to the modeling of a specific geographic area to a single processing unit in an effort to support load balancing. In this architecture a separate connection to the DIS network is provided for entity state PDU traffic and dead reckoning so that the portal's main function is to process jobs within the terrain manager. However, network communications is still a problem. In fact, this consistent bottleneck hinders the architecture's attempt to migrate jobs related to a specific geographic area to the unit handling that area. The single shared database problem described above also still exists and to a greater extent than in the previous architecture. Nevertheless, this architecture proved to be more scalable than the first and is better able to recover from system failures.

Finally, the developers sought to provide more flexibility in terms of implementation with their last architecture design. This design centers around the concept

of utility processes running in the background of all host computers in a DIS exercise and it is these processes called services that implement the shared environment.

The services and simulation application programs interact in a Client/Server manner in which the services are the servers and the applications are the clients. This design means that the vehicle simulators have the same ability to communicate with the shared environment as the physical modeling programs have. The services consist of the Entity Service and the Terrain Service.

The Entity Service is a single application that performs DIS networking, executes dead reckoning for remote sites, and supports multiple simultaneous client applications' requests for entity information. In addition, the Entity Service handles entity state, fire, and detonation PDUs. One copy of the Entity Service exists on each machine and thus, it acts as an intermediary between client applications (which each have their own channel to the service) and the DIS network. In addition, having this encapsulated service means that more than one application may be run on a machine.

The Terrain Service is a separate process that handles the state of the terrain. It contains an instance of the Dynamic Terrain Database and allows client applications to query and update this database. In addition, it handles the transmission and reception of the architecture specific dynamic terrain PDUs. One experimental PDU is used by the Terrain Service to notify a client of terrain changes for which the application must determine whether or not it has been affected by that change and if this is the case then it must resample. Although not yet implemented the designers feel that the Terrain Service is the place to conduct dead reckoning algorithms.

Overall, this architecture's noted features include the ability to make changes to the configuration of the Terrain Manager without having to be concerned with their effect on the client applications and the encapsulation of the shared environment which provides for easier development of physical modeling algorithms. The architecture runs on Silicon Graphics workstations, but is not capable of meeting the performance standards of the

distributed training systems. However, it is useful as a testbed for simulating vehicle/ environment interaction.

4. Dynamic Terrain Summary

IST has definitely created a high-fidelity dynamic terrain simulator. Walters' on the other hand, employed simple computationally inexpensive objects that are recognized by the viewer as the items they are meant to represent - Thorp's "70% solution"[THOR87] - but, they do not exhibit the kind of realism that IST has been able to model. However, it is the use of these icons that allow Walter's earthworks to be implemented in a 3D real-time simulator like NPSNET (without detracting from the simulator's performance); whereas IST's models are not capable of achieving this type of performance. Still another difference between the two similar features is that Walter's terrain models are not networked, but IST's earthworks are. Thus, after comparing the advantages and disadvantages of both dynamic terrain implementations, we made it our goal to network dynamic terrain effects in NPSNET as realistically as possible while still maintaining real-time performance.

C. EXPLOSIONS

1. Nash

David Nash implemented ballistic trajectories within NPSNET [NASH92]. Nash's work simulates the motion of the projectile from the moment it leaves the weapon (including the processes that occur inside the bore of the weapon which place the round in a spin as well as a forward motion and are largely responsible for determining its behavior in flight) until the time it explodes in air or collides with another object or the ground. The path of the projectile is realistically simulated due to the use of an object-oriented modified point-mass model developed by Lieske and Reiter [LIES66] in which forces including gravity, air drag, lift, equilibrium yaw, and Coriolis effect act on spin-stabilized projectiles. The terminal characteristics (events taking place at the end of the trajectory) of the munitions are also modeled by using the Naval Postgraduate School Explosions Editor (NPSEE). This editor enables a user to adjust various weapons and explosion environment

attributes and graphically view the results of the changes. These attributes are then saved to a file for use in an animation sequence. However, the animations are only of the explosions of the munitions (specifically chemical energy and shrapnel producing weapons), and not of the objects impacted by the weapons.

2. Monahan

James Monahan also provided a way to incorporate such physically-based modeling in three-dimensional simulations by enhancing the Naval Postgraduate School's Object File Format (OFF) [MONA91]. OFF is a standard set of files that contain object characteristics (such as material and light specifications) and routines that read in these attributes and allow for manipulation of the objects. The main purpose of OFF files is to provide a simple way to modify and/or port commonly used objects, attributes, and routines between various platforms without having to "reinvent the wheel every time." Monahan added the capability to create objects with various physical characteristics and constraints which are used to control the objects' motion due to internal and external forces acting on them. Forces are associated with each object in that a linked list of such forces are part of the overall programmed structure of the object (these objects are also maintained in a linked list of objects).

Forces modeled are both deforming (breaking or bending of the object) and non-deforming. Newton's laws are applied to calculate non-deforming force vectors which are then converted into two collision coordinate system vectors that affect torque and translation respectively. These vectors are then used to compute object frame movement values (acceleration and velocity) which are then mapped to corresponding world frame vectors and are ultimately used in modified Euler equations to determine the object's final position. Euler equations are used due to their simplicity and iterative speed especially since force updates are calculated one at a time rather than in parallel. Global forces are also applied but at each object's center of mass causing only linear acceleration.

Deforming forces on the other hand, exist only if such forces when applied are intense enough to break or bend a polygon of an object. A force that is strong enough to break a polygon is simulated by removing that polygon from the object list and replacing it with a list of smaller shards which are computed by cutting the polygon into triangles in a spiral like inward motion. On the other hand, if the force only causes the polygon to bend then again the polygon token is removed from the list and is replaced with a bendable polygon whose bending force is modeled using Hookes' law along with a spherical spring simulation.

In addition, Monahan incorporated further primitives and a layered set of object behavior control OFF extensions. The first layer allows for specifying the units of measure to be used and provides routines needed to make conversions so that OFF objects may be made compatible with any platform/simulator. The next layer provides the user with the ability to specify the object and force characteristics and constraints. And finally, layer three's purpose is to provide a mapping from an object's movement changes (via hardware input sources) to its afflicting forces' descriptions (thus altering their affect based on user input changes).

Monahan's OFF Mover Tool allows a user to test these features before incorporating them into a simulation. A designer can make a variety of object and force characteristics modifications and view the effect of these changes on the OFF object from all perspectives. When the user is satisfied with the object's behavior, the user has the option of saving the modified object so that it may be implemented in any simulation that employs the OFF library of object and force functions.

In addition to the OFF library enhancements, Monahan measured the cost of employing OFF physically-based modeling techniques in terms of frames per second based on objects affected by both deforming and non-deforming forces. Tests involved small, average, and large numbers of objects or polygons per object that are affected by small, medium, and large sets of forces run on a single processor Silicon Graphics 4D/240 VGX workstation. With just this single processor the simulations were able to maintain a rate of

no less than 8 frames per second and as high as 20 frames per second. It is especially important to note that when an object (of varying amounts of polygons) was affected by a small set of deforming forces (explosions), the frame rate ranged from 16 to 10 frames per second.

3. Explosions Summary

Although a variety of explosive effects have been created for NPSNET in the past, NPSNET has since incorporated Performer and other modeling tools (e.g., Multigen). Performer converts the models into a format that it is capable of displaying and therefore, an algorithm for creating explosive effects based on this format needed to be developed. In addition, previously simulated explosions were not networked. Thus, our aim was to provide networked explosions in the improved version of NPSNET.

D. SUMMARY OF EFFECTS RESEARCH

The effects research presented in this chapter all have added to the realism of training simulators; however as discussed there are many areas in which they can be improved or enhanced. We have therefore chosen to address several of these issues. The first of these issues deals with cloud modeling. Clouds have been modeled within real-time simulators; however, gridded wind vector data has not been included in these simulations. In addition, when flying through these clouds, the performance in frame rate dropped significantly enough to fall out of real-time limits. On the other hand, Corbin was able to model obscurants (smoke plumes) that could be flown through in real-time. Thus, it was our intent to incorporate real-time "fly throughs" and clouds affected by multiple wind vectors using Corbin's puff model. As far as dynamic terrain simulation is concerned, we felt that since fully recognizable real-time models of dynamic terrain existed in NPSNET (and IST's high fidelity earthworks were incapable of meeting the performance standards necessary of a distributed real-time simulator), that the networking of dynamic terrain in real-time was the next logical step. Thus, it was our goal to implement the Destructible Entity PDUs and then evaluate their impact on the performance of NPSNET using the real-

time dynamic terrain models already incorporated. And finally, as stated previously, since NPSNET was redesigned to use Performer, explosive effects previously modeled are no longer compatible with this simulator. Therefore, we deemed it necessary to design explosive effects that a Performer-based distributed synthetic environment such as NPSNET could simulate.

IV. CLOUDS

Atmospheric obscurants such as clouds greatly enhance the realism of a synthetic environment and thus aid in further immersing a user into the world. Clouds are vital to the effectiveness of the virtual battlefield as a training aid since they can affect the battlefield in several ways. These ways include obscuring visibility and impacting target detection and the operation of infrared sensors [LADS94]. It is therefore, imperative that such training systems accurately model clouds so that military units are receiving comprehensive training. As previously discussed, using the virtual battlefield for conducting military exercises saves a great deal of time (especially in terms of planning complex joint exercises) and money and significantly improves the serviceman's decision making skills. Simulating weather effects further adds to these advantages, by allowing training commanders to create weather conditions of their choosing. Such "what if" scenarios can constantly be incorporated and adjusted so that servicemen are well prepared to capably handle any meteorological occurrence during times of real crises. It is impossible to receive this kind of exposure to the multitude of weather occurrences in field training exercises. However, in order for exercise commanders to have the ability to create such environmental effects in the virtual battlefield, the simulation designer must accurately model the nearly infinite amount of atmospheric happenings including the wide variety of cloud characteristics.

A. CLOUD CHARACTERISTICS

A cloud's appearance is a reflection of the temperature and moisture content of its surrounding atmosphere [NAVA79]. Clouds are made of condensed liquid or frozen water particles that vary in size from about 2 to 30 micrometers. There exist as many as 300 or as little as ten such particles in a milliliter of air. Additionally, the total amount of water contained within a cloud varies greatly too - a single cloud might hold as little as 0.1 gram or as much as five or more grams of water per cubic meter. Due to such vast ranges of these attributes, clouds are found in an infinite variety of forms.

Generally speaking, clouds are classified by their height and how they are formed. This height characterization and the general categories of clouds are listed in Table 3 below.

Altitude over Middle Latitude (In Feet)	Name	Description	Composition
High Clouds 18,000 to 45,000	Cirrus	Maretails Wispy & Feathery	Ice Crystals
	Cirrostratus	High Veil, halo Cloud	Ice Crystals
	Cirrocumulus	Mackerel Sky	Ice Crystals
Middle Clouds 6500 to 18,00	Altocumulus	Widespread, cotton ball	Ice and Water
	Altostratus	Thick to thin, overcast; high, no halos	Water and Ice
Low-Family Clouds Sea Level to 6500	Stratocumulus	Heavy rolls, low, widespread. Wavy base of even height	Water
	Stratus	Hazy cloud layer, like high fog. Somewhat uniform base	Water
	Nimbostratus	Low, dark gray	Water/ice crystals
Vertical Clouds Few hundred to 65,000	Cumulus	Fluffy, billowy clouds. Flat base, cotton ball top	Water
	Cumulonimbus	Thunderhead, Flat bottom and lofty top Anvil at top	Ice (upper levels) Water (lower levels)

Table 3: Cloud Types and Associated Conditions After Ref. [NAVA79]

All clouds form as a result of air cooling below its dew point, the point at which the air's relative humidity is 100%. This decrease in temperature is usually due to adiabatic expansion. Adiabatic expansion occurs when an air parcel ascends and consequently expands due to air moving over terrain of increasing height or over air of greater density. Lifting of air due to mechanical turbulence or thermal instability also causes adiabatic expansion. In general, there are two classifications of air ascension that account for the two basic ways a cloud forms - condensation within rising air currents - cumulus formations - or condensation within layered air that is relatively free of vertical currents - stratus formation. However, it is the variations in these two processes that account for the variety of cloud forms that exist such as those listed in Table 3.

In addition, each cloud type does not develop due to only one set of conditions. For example a stratus layer may form when moist air is lifted by turbulence resulting in a nearly homogeneous, structureless layer. Stratus also forms due to a warm air mass flowing in over an elevated coast or when air is lifted up the slopes of a mountain. Due to the many variations of clouds brought about by so many environmental factors we felt it was necessary to design an interactive cloud modeler as a starting point for eventually modeling these variations within a real-time simulator like NPSNET.

B. CLOUD MODELER

The Cloud Modeler is a tool that enables a user to interactively create a cloud by using a previously constructed puff who's material attributes may be changed. The user may also designate how many of these puffs are to compose the cloud. The modeler uses both C++ and Silicon Graphics' Performer so that clouds designed using this tool may be easily ported into NPSNET since it was also written using this programming language and tool. Performer's low-level library libpr is used to build the puff since it handles Silicon Graphics architecture-specific tuning issues to "provide optimized rendering functions, state control, and other functions that are fundamental to real-time graphics" and also interfaces with the IRIS Graphics Library [SGIA94]. Additionally, this library provides optimized mathematical functions that are employed in this program as well.

1. Puff Characteristics and Construction

libpr provides geometry primitives called pfGeoSets that are a collection of like geometry (lines, points, triangles, quads). Every pfGeoSet is made of only one type of primitive and all primitives part of that pfGeoSet share the same attribute bindings (texture, color, and normal bindings). The puff is made of one pfGeoSet containing three perpendicularly aligned 8-sided polygons. Each polygon is built using tristrips (each tristrip is considered to be a primitive) as illustrated in Figure 1 where the vertices' numbers represent the order in which the polygon is drawn. Tristrips is a very fast method of drawing objects since after the first three vertices are drawn to form the first triangle, only one line

is needed for each connected triangle following the first because the last two drawn vertices of the previously drawn triangle are shared with the next one to be constructed (triangle 1,2,3 shares vertices 2 and 3 with triangle 2,3,4 which shares vertices 3 and 4 with triangle 3,4,5, and so on). Sharing of vertices also allows for substantial savings of memory space.

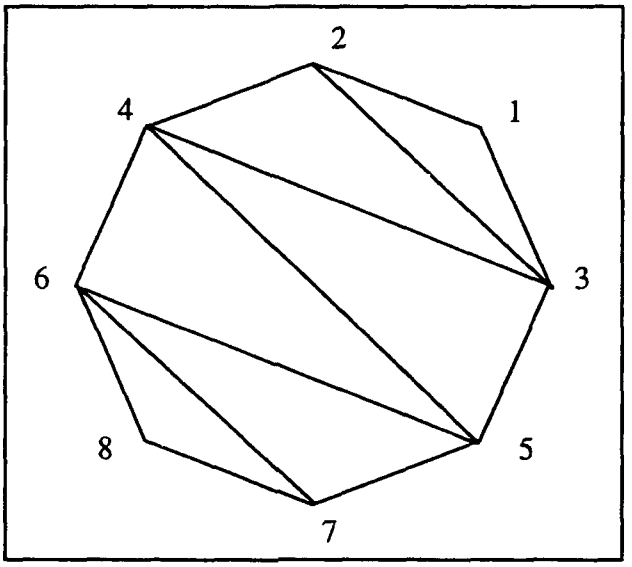

Figure 1: 8-sided Polygon Built Using Tristrip

In addition, normals are assigned per polygon of the puff as are texture mapping coordinates so as to completely cover each of the three polygons. However, further properties of the puff are assigned by associating a pfGeoState with the pfGeoSet.

"A pfGeoState is a structure that encapsulates all the graphics modes and attributes that libpr manages." [SGIA94] By setting various elements of a pfGeoState a graphics context is defined that may be applied to a pfGeoSet thereby completely defining both the geometry and graphics state. Any rendering attributes (such as material and light definitions) or modes (such as enabling texturing and lighting) not specifically assigned to the pfGeoState are inherited from the current global state by the pfGeoState.

In the case of the cloud puff, a variety of modes and attributes are defined and assigned to the pfGeoState which is assigned to the puff pfGeoSet. Texturing, transparency,

and lighting modes are all enabled so that these attributes may be used to characterize the puff. Material properties which define reflectance characteristics of surfaces including alpha (transparency), ambient, and diffuse color values are specified. Ambient reflectance is the appearance of the surface due to the scattering of lighting in the scene where as the diffuse property is the reflectance of a surface due to a directed light in the scene [ZYDA90]. Both reflectances occur uniformly in all directions and their values are assigned using values between 0.0 and 1.0 for the red, green, and blue (RGB) components. Finally, the alpha material property allows transparency to be specified and the amount is defined using numbers between 0.0 and 1.0 where 0.0 is totally transparent and 1.0 is completely opaque. The texture of the puff is applied by loading an RGBA file and clamping (only one image of the texture is applied to each polygon using the provided texture coordinates described above) it to each polygon. In addition, a texture environment is created to specify how the texture is to blend with the underlying colors of the polygons. Performer's PFTE_BLEND which is the same as GL's TV_BLEND token value is employed so that total blending occurs between the puff's polygons' materials including the alpha value and the texture's properties.

2. Lighting

The pfGeoState associated with the puff pfGeoSet is also assigned lighting characteristics so that the cloud's reflectance characteristics can vary as the light in the scene varies. This capability of the modeler is extremely important, because lighting significantly affects the obscuration of a cloud. A cloud can appear to an observer totally translucent when no light is shining on it, but opaque if a light is shining on it at just the right angle [BOEH94].

The lighting created for this model consists of a light along with a light model. The light is designated as an infinite light (as opposed to a local one) since this is how the sun, the main light that affects clouds, is categorized. In addition, the light provides ambient lighting based on the RGB color values that this component of the light is assigned by the

user and it emits light whose color is also specified using RGB components. The puff's ambient reflectance and diffuse reflectance are dependent on the amount and color of ambient and diffuse light provided by the sun model respectively. Although the light is infinite, the direction from which the light comes from is provided and may be manipulated.

The light model on the other hand, must be specified so that the lighting just described may take effect. The light model is used to specify the amount of total ambient light present in the entire scene (not just from the sun) and to enable two-sided lighting. This model also provides the capability to select attenuation calculations desired, but since the modeler only has an infinite light, attenuation does not play a role in this simulation.

3. Resulting pfGeoState Associated With the Cloud Puff

After all of the above attributes and lighting are assigned to a pfGeoState this pfGeoState is assigned to the puff pfGeoSet and the initial puff is ready to be drawn. As a result of this assignment, every time the puff is drawn by Performer the associated pfGeoState is applied to the geometry pipeline so that the attributes set in the pfGeoState are applied to the pfGeoSet's appearance. Example Performer code used to define and assign rendering modes and attributes to a pfGeoState and subsequently attach it to the puff pfGeoSet is illustrated in Figure 2.

4. Adding the Puff to the Scene Graph Hierarchy

After the construction of the initial puff, the Cloud Modeler places the pfGeoSet into a run-time scene graph hierarchy. The scene hierarchy maintains state information and geometry, and defines how items in the database relate to one another [SGIA94]. Various connected node types each with a specific function compose this directed acyclic graph. Connection among nodes allows for inheritance and thus sharing of database units. The graph is usually constructed by the application at load time and during execution it is continually culled (by traversing the graph using a depth-first search) resulting in all visible geometry being added to the display list which is then rendered by the draw process. Table 4 below displays the nodes used by the Cloud Modeler.

```
//*****Create a new pfGeoState from shared memory and enable
//*****texturing, transparency, and lighting.
gst = pfNewGState(arena);
pfGStateMode(gst, PFSTATE_ENTEXTURE, 1);
pfGStateMode(gst, PFSTATE_TRANSPARENCY, 1);
pfGStateMode(gst, PFSTATE_ENLIGHTING, 1);
//*****Set up initial materials for puff and assign to pfGeoState, gst.
mtl = pfNewMtl(arena);
pfMtlColorMode (mtl, PFMTL_BOTH, PFMTL_CMODE_COLOR);
pfMtlColor(mtl, PFMTL_AMBIENT, rm_amb, gm_amb, bm_amb);
pfMtlColor(mtl, PFMTL_DIFFUSE, rm_diff, gm_diff, bm_diff);
pfMtlAlpha(mtl, transp);
pfGStateAttr(gst, PFSTATE_FRONTMTL, mtl);
pfGStateAttr(gst, PFSTATE_BACKMTL, mtl);
//*****Set up texture (and environment) and assign to gst.
tex = pfNewTex(arena);
pfLoadTexFile(tex,"puff.rgba");
pfTexRepeat(tex, PFTEX_WRAP, PFTEX_CLAMP);
tenv = pfNewTEnv(arena);
pfTEnvMode(tenv, PFTE_BLEND);
pfGStateAttr(gst, PFSTATE_TEXTURE, tex);
pfGStateAttr(gst, PFSTATE_TEXENV, tenv);
//*****Create a light source coming from the "south-west" and a lighting
//*****model and assign both to gst.
SunArray[0] = pfNewLight(arena);
pfLightColor(SunArray[0], rl_col, gl_col, bl_col);
pfLightAmbient(SunArray[0], rl_amb, gl_amb, bl_amb);
pfLightPos(SunArray[0], x_pos, y_pos, z_pos, 0.0f);
lm = pfNewLModel(arena);
pfLModelAmbient(lm, rlm_amb, glm_amb, blm_amb);
pfLModelTwoSide(lm, TRUE);
pfGStateAttr (gst, PFSTATE_LIGHTS, SunArray);
pfGStateAttr (gst, PFSTATE_LIGHTMODEL, lm);
//*****Attach pfGeoState, gst, to the puff pfGeoSet, gset1.
pfGSetGState(gset1, gst);
```

Figure 2: Creating a pfGeoState to Attach to pfGeoSet

Node Type	Node Class	Description
pfScene	Root	Parent of visual database
pfSwitch	Branch	Selects among multiple children for displaying
pfDCS	Branch	Dynamic Coordinate System - used to move scene objects
pfGeode	Leaf	Contains geometric specifications (pfGeoSets)

Table 4: Applicable Iris Performer Node Types After Ref. [SGIA94]

In order to incorporate the puff into the scene hierarchy, the puff's pfGeoSet is first added to a pfGeode node. The graph is then built by starting with the pfScene node as the root, adding pfSwitch nodes as branches (children) to the pfScene, extending a pfDCS node from each of the pfSwitches and finally connecting the pfDCS nodes to the one pfGeode containing the puff. The purpose of the switches is to add further functionality to the modeler. They allow a user to model a cloud with more than one puff by "turning on and off" children extending from these nodes. Using a pfSwitch node to turn off a child tells the cull process not to include that node in the display list to be rendered for that frame. The pfDCS nodes provide the capability to scale the puff and provide support for user selected distributions of multiple puffs in that they are used to translate the puffs to their proper position within the chosen distribution. Distribution options are described in Section 6. The cloud modeler's scene graph is displayed in Figure 3.

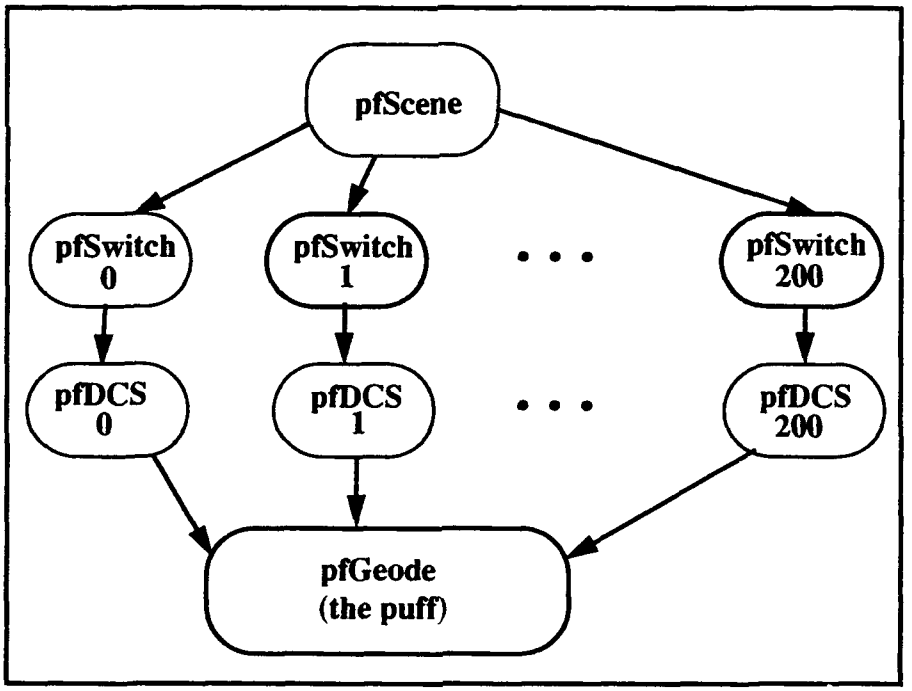

Figure 3: Cloud Modeler Scene Hierarchy

An interesting feature of this graph is that there is only one puff node (pfGeode) even though the cloud modeler has the ability to display many puffs at one time. Only one

node is needed in this case, because Performer provides a memory saving capability called shared instancing [SGIA94]. Shared instancing involves adding a node to two or more parents thereby giving each parent its own copy of that node. However, changes (attribute changes) made to the node itself, not a copy, are reflected in all copies. On the other hand, when instancing is incorporated, changes made to a parent node are propagated down to its children, and if any of its children is a copy of an instanced node then only that parent's copy is affected. The cloud modeler's scene graph in Figure 3 incorporates both of these inheritance properties. For example, all of the pfDCS nodes share the one pfGeode puff and when a change is made to the puff's alpha value via the puff's pfGeoState, all of the pfDCS nodes' local copies reflect this modification. However, if pfDCS node 0 is translated 30 meters in the positive X direction then its child is also translated the same amount. This movement does not affect any of the other copies of the puff. In addition, each pfDCS node has a pfSwitch node as its parent so changes made to the pfSwitch node's child and/or made to the child of this child are experienced by the pfSwitch node. Furthermore, if a pfSwitch is "turned off" then so are its children - specifically the copy of the puff extending from this pfSwitch via the pfDCS is not rendered. These examples and the overall concepts of inheritance are vital to the functionality of the cloud modeler as is better explained in the next section.

5. Cloud Modeler's Capabilities

As previously stated, once the initial puff is created and added to the scene graph, the user is free to change any of the many parameters defining the puff or puffs. The modeler displays one puff initially, but as mentioned the user may add more puffs to the scene. This feature and the many additional options available to a designer are listed below with further explanation following the list.

- Change the diffuse RGB material properties of the puff(s) in increments or decrements of 0.1.
- Change the ambient RGB material properties of the puff(s) in increments or decrements of 0.1.

- Increase or decrease the transparency by 0.1.
- Scale the puff size by factors of 10, 20, 30, 40, 50, 60, 75, 100, 125, & 150.
- Input the desired number of puffs to make up the cloud (1, 2, 3, 30, 40, 50, 75, 100, 150, & 200).
- Change the scene's light's ambient components (RGB) in increments or decrements of 0.1.
- Change the scene's light's color components (RGB) in increments or decrements of 0.1.
- Change the direction from where the light is coming - northwest, southwest, northeast, or southeast.
- Change the scene's light model's ambient components (RGB) in increments or decrements of 0.1.
- Examine the texture applied to the polygon by itself without material properties contributing to its appearance.
- Choose a desired distribution and depth of puffs - uniform, normal, or a symmetrically normal distribution about the X-axis.

The above options are manipulated by the designer interactively by pushing a set of keys as described in Appendix A, the Cloud Modeler User's Guide, and the program responds to these key punches by calling the appropriate callback function which adjusts the selected attribute within the puff's pfGeoState. In addition, these parameters' values are displayed on the screen at all times. Furthermore, the program provides a moving eyepoint by way of the mouse which allows the user to thoroughly examine the cloud (cloud is stationary) from its exterior and interior and from all angles.

The capability to have many puffs arranged in designated distributions requires further explanation. When a user requests that more than one puff be displayed in the world, the program responds by placing the amount designated on the screen in a horizontal row. More or less puffs, depending on the user's request, are displayed in the scene by turning on or off the pfSwitch nodes and each pfDCS node is used to translate each copy of the puff to make this row. Once the designer selects the amount of puffs to make up the cloud, a distribution of these puffs may also be selected.

6. Cloud Puff Distributions

The following distributions are provided by the modeler: uniform, normal, and symmetrical normal - an arrangement that is the same as a regular normal distribution except that another inverted normal distribution is present which is symmetrical to the first about the X-axis. All three distributions also accept a user specified depth and the uniform distribution allows the user to select how many rows of puffs are to be displayed. An increase in depth increases the opacity of the cloud. There are however, limitations on depth requests - there must be at least 20 puffs to apply a depth of three, 50 puffs to apply a depth of five, and 100 puffs to apply a depth of ten. Normal and symmetrical normal distributions are used to provide a "fading" affect on the outer edges of the cloud. In addition, all of these commonly known "organized" distributions are incorporated so that a designer may take a look at a cloud and relatively easily (due to familiarity with the distribution at hand) determine adjustments that will produce cloud formations needed for modeling atmospheric effects desired.

a. Uniform Distribution

The uniform distribution employs the simplest algorithm of the puff arrangements available. Once a user specifies a depth and the amount of rows that the cloud is to have, the uniform distribution algorithm builds a cloud from the left and up meaning that it starts at the left most position, applies the required depth, then moves up to the next row and applies the same depth and continues in this manner until the required number of rows are present at the leftmost X position. At this point, the algorithm moves right and down to the first row and continues with the same algorithm until the cloud is built. All puffs are equally spaced by using their bounding volumes. After a puff is positioned, the next one is placed at the proper row, depth, and column but relative to the previous puff's bounding volume so that they are partially overlapped which avoids displaying unrealistic holes. This method of construction is illustrated in Figure 4. In addition, Figure 5 is a

display of a cloud composed of 100 uniformly distributed puffs in four rows with a depth of three as it appears on the computer screen.

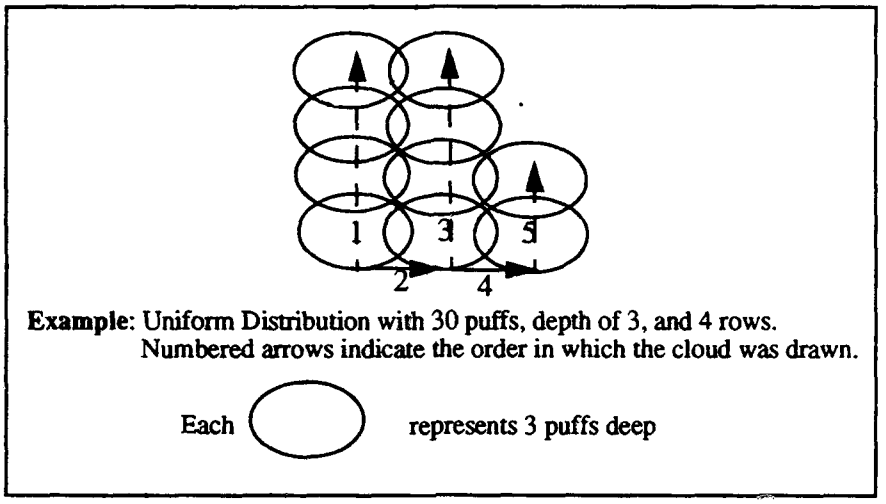

Figure 4: Uniform Distribution Drawing Order

b. Normal Distribution

The normal distribution used in the Cloud Modeler is a mathematically correct approximation of a standard normal distribution. The approximation method is based on the fact that **most** of the area under a standard normal curve (a curve symmetric about the vertical axis through the origin of a plane) lies between -3 and 3 of the horizontal axis [WEIS91] as illustrated in Figure 6. The numbers labeled inside the curve indicate the fraction of the area of the normal curve that exists between two consecutive whole number boundaries. When all of the fractional numbers are added together they account for 99.74% of the area of the normal curve. Therefore, when a normal distribution is selected by the user, the Cloud Modeler models this distribution by distributing the total number of puffs in the scene among six areas (or columns) just as Figure 6 illustrates. Distributing the puffs in this way requires that the algorithm multiply the total number of puffs in the scene individually by each of the fractional numbers in Figure 6 and use the results to fill each of the cloud's six single-puff wide columns. Additionally, depth is incorporated in the same

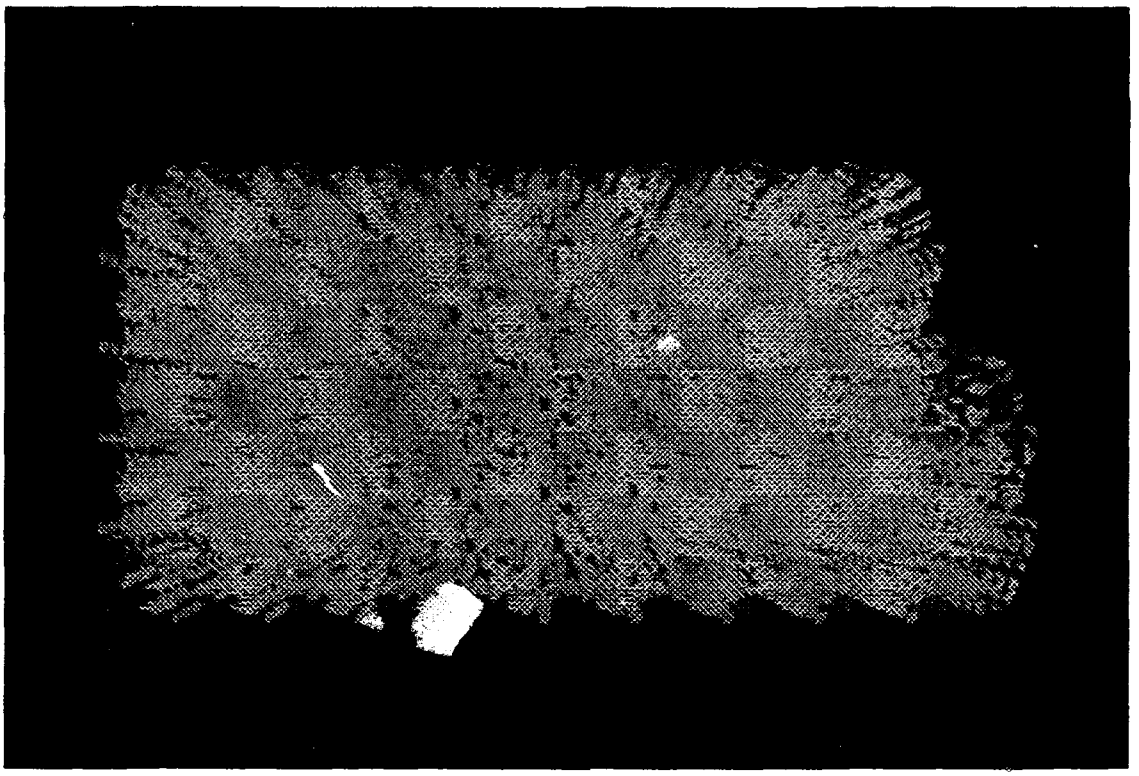

Figure 5: Uniformly Distributed Cloud - 100 puffs, 4 Rows, and Depth of 3

way that it is in the uniform distribution. The cloud is initially constructed from the bottom left and built up until 0.0215 of the total puffs have been placed in the leftmost column. During this build up, depth is incorporated at each level of the column (row) before the algorithm places a puff(s) in the next row. For example if the column is to have six puffs in it and the depth selected is one then there will be six rows of one puff in that region; whereas if a depth of three is requested then there will only be two rows with three puffs in each (extending back). The algorithm continues to the next column on the right and builds in the same manner as with the previous column. This overall routine continues until a normal distribution is represented by six columns of puffs. Figure 7 contains a cloud composed of 100 normally distributed puffs with a depth of ten.

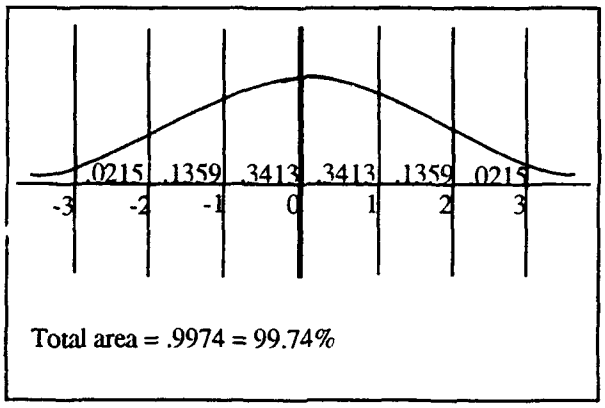

Figure 6: A Standard Normal Distribution Curve

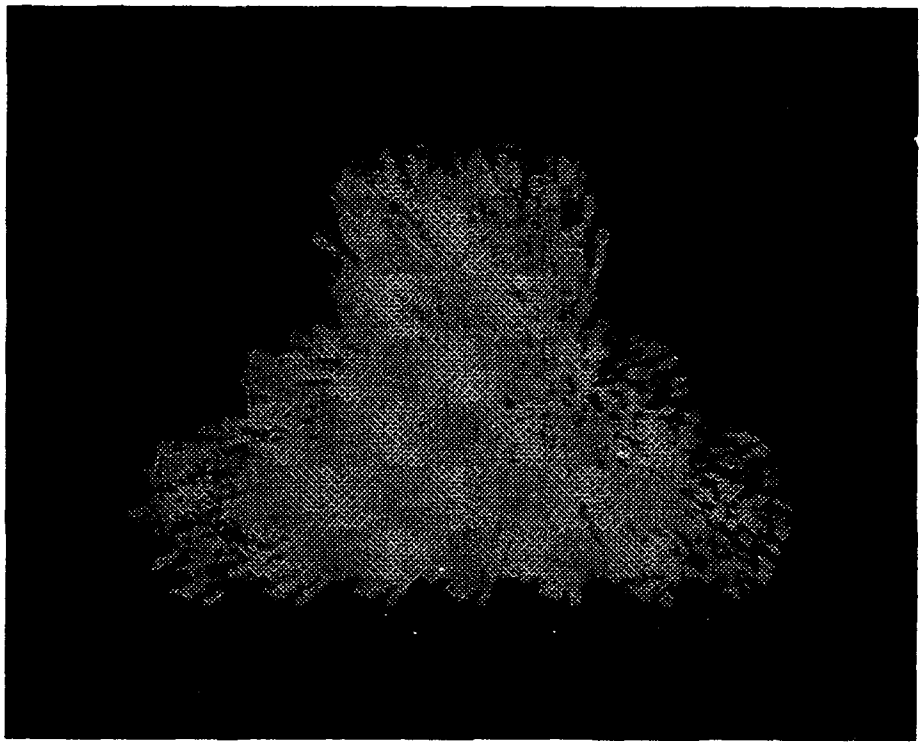

Figure 7: Cloud With Normal Distribution of 100 Puffs and Depth of 10

c. Symmetrical Normal Distribution

Since the standard normal distribution often results in very high middle areas of puffs, we decided to also incorporate the symmetrical normal distribution option.

This option still provides for fading along the right and left edges of the cloud, but also distributes the puffs lower along the vertical axis for a more uniform distribution as illustrated in Figure 8. The algorithm for this arrangement of puffs is exactly the same as the regular normal distribution except that it alternately builds rows above and below the imaginary X-axis starting with above the axis. In other words, the program builds one row (with one puff or multiple puffs if a depth has been selected) and then if there are enougn puffs for this column, the next row is built on the opposite side of the X-axis and this alternating of sides continues throughout the building process of each of the columns. Figure 9 illustrates a cloud composed of a symmetrical normal distribution of 100 puffs with a depth of ten.

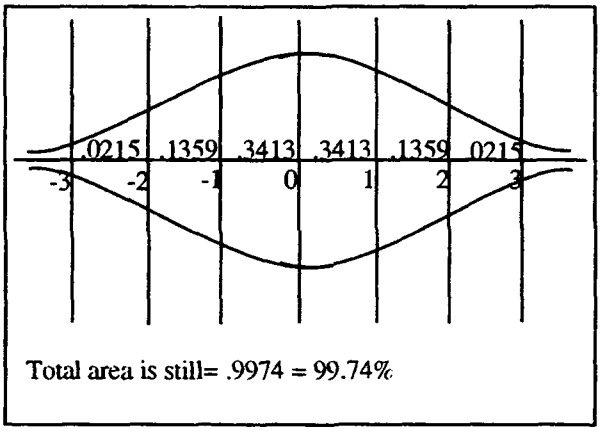

Figure 8: The Symmetrical Normal Distribution

7. Performance of and Applications for the Cloud Modeler

In order to determine the effect that clouds modeled with this program might have on the frame rate of a real-time simulator we ran a series of tests within the modeler on a Silicon Graphics Reality Engine. Initially we found that the distribution used did not affect our results; however, the depth did. Therefore, we modeled four different clouds each composed of a uniform distribution of puffs with varying depths and evaluated the frame rate (against a requested frame rate of 20 frames per second) of all four from a still

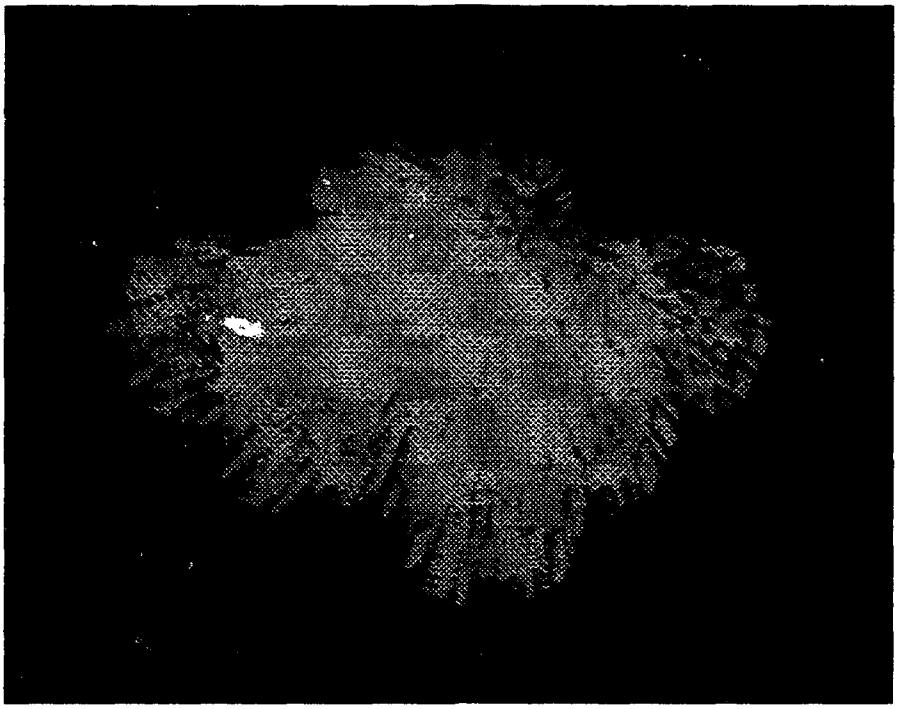

Figure 9: Cloud With Symmetrical Normal Distribution of 100 Puffs and Depth of 10

viewpoint where the entire cloud was in the viewport and then drove through the middle of the cloud. The results of these tests are displayed in Table 5. We concluded that clouds with over 50 puffs cannot be displayed completely at the minimal real-time frame rate of 10 frames per second even when the user is not moving. Driving through the clouds results in much worse performance and as illustrated in Table 5 the frame rate decreased to as little as 2.0 frames per second due to the increased depth that the user had to drive through.

Number of Puffs	Rows	Depth	Still Viewpoint Frame Rate (frames per sec.)	Worst Case Frame Rate - Driving Through Clouds
1	1	1	20.0	10.0
50	3	3	10.0	4.0
200	5	5	5.0	2.5
200	4	10	6.7	2.0

Table 5: Performance Tests of Clouds Composed of Uniformly Distributed Puffs

As a result of these tests, we felt it was necessary to provide a way for users to evaluate the performance of clouds that they designed with the Cloud Modeler in their own simulators. We added a feature to the modeler that allows a designer to save the parameters of any cloud modeled to a file to be read by a simulator aware of how the cloud information is stored. The file contains current values for all of the parameters that the user is allowed to modify. In addition, we developed a program that reads this file in order to place clouds in its world of wind vectors. This simulator and its performance is described in Section D of this chapter.

Another application of this modeler is for modeling a uniform stratus layer. A layer has been incorporated in the modeler that is separate from the puff and the rest of the scene hierarchy. The layer's characteristics of the height of both the top and bottom, thickness, color, and transition zones between the sky and the start of the top and bottom of the layer are all adjustable. In addition, this model enabled us to experiment with various stratus layer appearances and then place one in NPSNET that too can be adjusted. Concepts regarding the stratus layer and its implementation are discussed in the next major section of this chapter.

C. MODELING THE STRATUS LAYER

1. Characteristics

As stated in Section A of this chapter, a stratus layer is a sheetlike cloud that develops as a result of condensation within layered air that is relatively free of vertical currents. These layers are part of the low-family cloud classification (which includes stratocumulus and nimbostratus clouds as well) that range in altitude from 1980 meters (6500 feet) to just above the ground [NAVA79]. Strictly speaking stratus clouds are layers of fog or clouds that are not far above the ground or sea (500 meters or 1600 feet high) and they are basically formless. The difference between fog and a stratus layer is that the stratus layer does not reduce horizontal visibility at the surface below it [FAA65]. However, fog and stratus many times exist together, but there is really no firm distinction between the two

other than the height at which they are found and their difference in denseness (stratus layer is more dense).

In general, figures that are used to describe stratus layers other than their height as stated above are their thickness and their horizontal dimension. A stratus layer is typically 0.5 to 1 kilometer thick and its horizontal dimension may range from 10 to 1000 kilometers (6 to 600 miles). In addition, the total area encompassed by a stratus layer is as large as 1,000,000 square kilometers [SCHA81].

2. Stratus Layer in the Cloud Modeler

Based on these stratus characteristics, we added a stratus layer to the cloud modeler. The stratus was incorporated by employing a set of Performer's functions that are used for creating environmental visual effects (sky, horizon, ground, clouds, and fog), called pfEarthSky [SGIA94]. Performer incorporates approximations of visibility effects by using a multiple-layer sky model as illustrated in Figure 10. The fog layer extends from the ground to the area of general visibility thinning out as its altitude increases. The general visibility area extends to the bottom of the lower transition zone of the cloud layer if in fact the programmer has elected to incorporate clouds and the optional transition zones. These zones provide a smooth transition between the area of general visibility and the opaque layer of near-zero visibility that represent the clouds. If clouds do not exist then the area of general visibility extends infinitely from the boundary of the fog layer or ground.

As far as the cloud functions are concerned, Performer allows a designer to specify the height of the top and bottom of the cloud layer, the colors of both the top and bottom in RGBA components (different top and bottom colors result in a blending effect, but currently the alpha component has no effect), and the height of the top and bottom transition zones. Therefore, our cloud modeler allows the user to interactively modify these parameters and examine their effect on the stratus layer. In addition, the user is also able to control the thickness separately. In other words, changing the layer's top or bottom boundaries results in the opposite boundary's height being altered by the same amount so

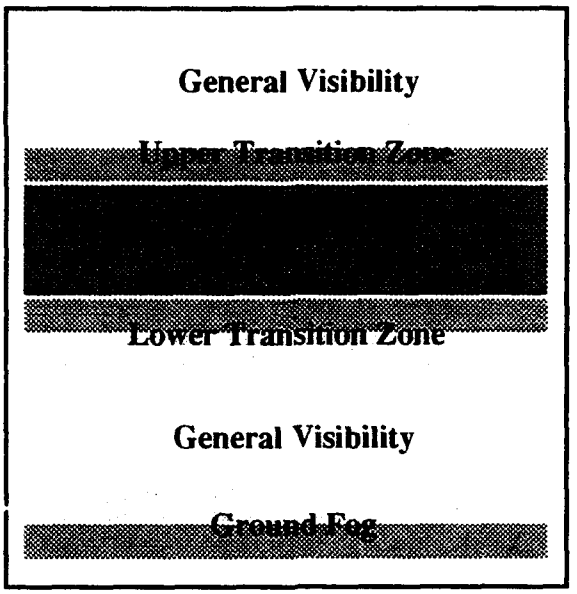

Figure 10: Performer's Layered Atmospheric Model From Ref. [SGIA94]

that the thickness of the layer remains constant. In order to increase or decrease the thickness of the stratus a separate key punch sequence is provided. When this feature is selected, the thickness algorithm equally distributes the thickness change between the top and bottom boundary heights meaning if the thickness is increased by 100 meters then the height of the top boundary is increased by 50 meters and the height of the bottom boundary is decreased by 50 meters. This feature along with the others described related to the stratus layer allow the user to fully take advantage of Performer's cloud layer capabilities and provide the designer with the capability to model sheet-like clouds for eventual use within Performer-based simulators like NPSNET as we have done.

3. Stratus Layer in NPSNET

Our implementation of the low level stratus cloud within NPSNET involves providing the user with the capability for turning on or off the cloud layer, changing the top and bottom boundaries as described above and adjusting the thickness. There are however, limitations on all of these features. Cloud parameters are only allowed to be changed if the clouds are visible. When adjusting the heights of the boundaries the user is limited to 1980

meters (6500 feet) and 500 meters (1600 feet) for the top and bottom of the layer respectively. In addition, the minimum and maximum thicknesses allowed are 500 meters and 1000 meters respectively and are incorporated using the same algorithm as the modeler with one exception - if the algorithm described above for increasing or decreasing the thickness generates a boundary value that is beyond that boundary's limit, then this boundary is placed at its limit and a greater amount is added or subtracted from the other boundary in order to model the thickness requested. This idea also pertains to increasing and decreasing the boundaries' heights in that if an increase or decrease were to cause a boundary to extend over its limit then it is moved to its limit and the other boundary is placed the thickness value away from the limited boundary. Finally, both of the transition zones have constant values, each start 150 meters from the corresponding boundary, and are automatically adjusted to their proper positions every time the heights of the top and bottom of the cloud layer are modified.

Overall, the limitations selected correspond to those of typical stratus layers as described above in the Characteristics section. The pfEarthSky functions along with these parameters provide realistic modeling of the layer from below the stratus and within it and do not detract from NPSNET's real-time performance.

D. CLOUDS AFFECTED BY WINDS

Since clouds are not stationary, but rather are affected by the movement of the air mass that surrounds them we felt it was necessary to construct a "proof of concept" world of wind vectors that affect the movement of the clouds. As, Nash states in [FAA65], "As the transportation agency for water vapor, wind has an important effect on the formation of fogs and clouds and on the production of precipitation." A description of this effect on the formation of clouds is found in the Cloud Characteristics section of this chapter. And thus, since it is the air currents that cause the clouds to develop in the first place, it is imperative that winds be considered when modeling clouds.

Our wind vector Performer-based testbed is composed of a 10,000 meter by 10,000 meter world divided into ten cells, each containing a different wind velocity and heading as displayed in Figure 11. The cells are 2000 meters by 5000 meters. A moving eyepoint via a mouse and a number of key punches allow the user to view the world from a variety of perspectives.

Cell 0: Vel: 5 m/s Head: 5°	Cell 1: Vel: 25 m/s Head: 5°	Cell 2: Vel: 5 m/s Head: 20°	Cell 3: Vel: 5 m/s Head: 160°	Cell 4: Vel: 20 m/s Head: 180°
Cell 5: Vel: 10 m/s Head: 0°	Cell 6: Vel: 10 m/s Head: 30°	Cell 7: Vel: 10 m/s Head: 170°	Cell 8: Vel: 10 m/s Head: 135°	Cell 9: Vel: 35 m/s Head: 180°

Figure 11: Winds Within the Wind Vector Simulator

Clouds are inserted in the world initially by reading a file of parameters produced by our Cloud Modeler and modeling the clouds specified using basically the same modeling algorithms incorporated within the modeler. Figure 12 illustrates the scene hierarchy incorporated in this simulator. Again, one pfGeode node is copied so that the puff is the child of many pfDCS nodes; however, unlike the Cloud Modeler's scene graph, a pfSwitch is included for each cloud and not per puff. This use of the pfSwitch node

provides a simple method of grouping puffs according to what cloud they belong to and is useful when the cloud exits the boundaries of the world as is explained below.

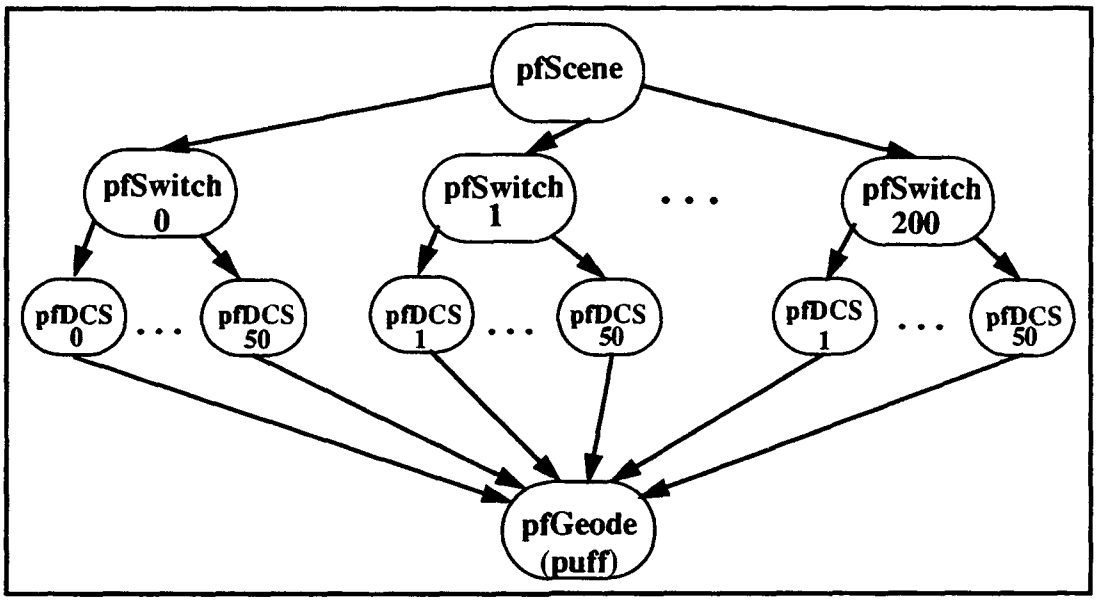

Figure 12: The Scene Hierarchy for the Wind Vector Simulator

The wind vector simulator unlike the Cloud Modeler imposes a 50 puff per cloud limitation on clouds that may exist in its world. This restriction is enforced by using a default cloud model puff arrangement anytime the puff amount obtained from the input file is over 50. The default arrangement consists of five puffs arranged as illustrated in Figure 13. However, this default cloud still incorporates the rest of the file's parameters in each of the 200 clouds that are placed in the world. If, on the other hand, the amount of puffs specified in the modeler output file is less than or equal to 50 then 50 clouds modeled using the distribution indicated are placed in the wind vector world. Fewer clouds are incorporated in this situation in an effort to limit the total amount of puffs present in the world.

Once, placed in the world, the clouds are subjected to the wind vectors of the cells within which they exist. The program determines which cell the cloud is in by comparing the position of the middle puff of the cloud with the cells' boundaries. Each cloud moves

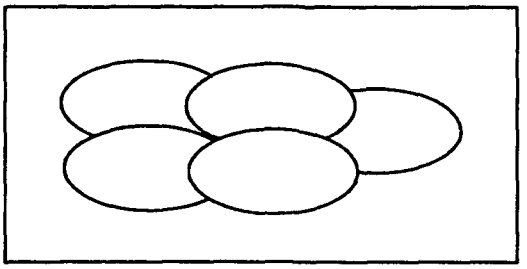

Figure 13: Default Cloud

at the velocity and in the direction associated with the wind of the cell meaning that every puff that is part of the same cloud is moved exactly the same amount in the same direction. This movement is calculated each frame using the following common vector component equations (delta_time is incorporated to calculate movement in meters per second).

$$x_position = \cos(\phi) \times velocity \times \cos(\alpha) \times delta_time \quad \text{(Eq 4.1)}$$

$$y_position = \cos(\phi) \times velocity \times \sin(\phi) \times delta_time \quad \text{(Eq 4.2)}$$

$$z_position = \sin(\alpha) \times velocity \times delta_time \quad \text{(Eq 4.3)}$$

ϕ = the cloud's pitch

α = the cloud's heading

As the clouds move through the world they enter new cells and are thus, made to respond to the new wind vectors. However, a cloud does not immediately start moving at the same speed as the wind. It accelerates or decelerates depending on the change that is experienced. The acceleration and deceleration algorithms are not physically-based, but rather provide the appearance of a gradual increase or decrease in speed until the cloud obtains the speed and heading of the wind in its cell. Likewise, when a cloud leaves the boundaries of the world - 0 to 10,000 meters in X, 0 to 10,000 meters in Y, and greater than 4000 meters in Z - the cloud is "turned off" using the pfSwitch node, but is immediately turned back on again at one of the two X boundaries (at a randomly chosen Y and Z position) so that there is always the same number of clouds in the world (either 50 or 200).

The winds themselves were chosen to demonstrate definite changes in the clouds' movement in order to verify that indeed the clouds are moving as they are supposed to. For example, a cloud that moves from cell 9 (35 m/s and 180^{o}) to cell 8 (10 m/s and 180^{o}) decelerates from 35 to 10 meters per second. In addition, opposing winds were are also implemented to demonstrate accurate cloud movement. A part of the world where clouds experience opposing winds is when they move from cell 2 (5 m/s and 20^{o}) to cell 3 (5 m/s and 160^{o}) or vice versa. At this border they are constantly being pushed from the one cell to the next and thus, end up moving upward until they reach the height boundary of 4000 meters at which point they die and are brought back into the world at one of the two X boundaries.

Coupling this variety of winds with the multitude of cloud formations causes performance problems as we found when running a series of performance tests. These tests involved using the default, a normal distribution, and symmetrical normal distribution cloud. Using the default cloud meant that there were 1000 puffs in the world at one time. When viewing no more than three cells at a time of the world filled with these clouds the frame rate was on average 6.7 frames per second; but when the user's view was enlarged the frame rate dropped to 4.0 frames per second. The frame rate for an aerial view of the entire world was also 4.0 frames per second. The normal distribution cloud used was composed of 50 puffs with a depth of five which means there were 2500 puffs in the world at any one time. The frame rate for this type of cloud varied from 1.4 frames per second when looking at the aerial view to 4.0 frames when there were only four to five clouds in view. Finally, the cloud composed of puffs arranged in a symmetrical normal distribution was made of 40 puffs with a depth of three meaning that there were 2000 puffs in the world. The frame rates were just about as low as the previous cloud type's - 5.0 frames per second when six to seven clouds were in view and 1.8 to 1.9 frames per second when viewing the entire world from the air.

The frame rates obtained were not a surprise considering those that had been previously obtained from the Cloud Modeler. However, they did demonstrate the affect that the total number of puffs in the world had on the operation of the simulator no matter where the viewer was looking. In addition, we found that even though the default cloud had almost no depth to it, the wind vector simulator still was not able to run in real-time most likely due in part to the movement calculations of both the clouds and the viewer, use of texturing and lack of multiprocessing.

E. CLOUD SUMMARY

Overall, we found that we are able to model fairly realistic looking clouds; however as they stand, they cannot be implemented in real-time and especially not when they are moving. Nonetheless, the Cloud Modeler is a tool that enables us to model what we think might work in a simulator and then "try it out." This strategy worked with incorporating the stratus layer into NPSNET. The wind vector simulator again does not run in real-time with the Cloud Modeler's clouds, but provides a testbed for wind vector simulating - not only for simulating how the clouds react to winds, but for testing how other atmospheric objects react as well.

V. NETWORKING DYNAMIC TERRAIN

Modeling dynamic terrain is crucial to simulating a realistic synthetic battlefield environment. Terrain changes play an important role in decision making during times of war in such circumstances as considering the idea of strategically constructing a berm(s) in order to slow the enemy's pursuit or building or destroying a bridge in an effort to get to another part of the land or to stop an enemy's advance respectively. In addition, craters which impede the advancement of troops generously populate a war stricken area due to fires/detonations that miss the intended target and strike the ground instead or due to a unit destroying no longer needed protective berms. The above examples of war time modifications to the terrain illustrate the importance of incorporating realistically modeled terrain modifications within virtual battlefield simulators.

As previously described, such features have been incorporated in 3D DIS simulators such as NPSNET; however, networking of the state of the terrain has yet to be implemented using a standardized DIS protocol data unit (PDU) format. Networking is essential to virtual battlefield simulations since this tool allows many players to interact with the world simultaneously from distant areas; thereby providing remote military units the opportunity to train together without having to relocate to a common location. Thus, when players are interacting over a distributed system it is imperative that all changes occurring in one player's world also be reflected in all other player's worlds. Networking protocols provide support for this homogenous world. However, networking is a precarious issue since a designer of a networked simulator must ensure that only the necessary information is communicated to the hosts so as not to hamper the networks's real-time performance; while at the same time ensuring that enough non-system specific data is provided so that any platform regardless of its hardware properties or limitations is able to understand and reproduce an object's appearance, geometry, and placement [WALT92]. Furthermore, another important feature that distributed systems must incorporate is a recording process in order to provide update information (send previously transmitted state

changes) to late joining hosts or in the case of a system failure, to have access to the state of the simulation previous to the interruption so that it may be restarted at that state. Thus, we sought to implement a recording and transmission process for the state-of-the-world [ZYDA93] using the PDUs developed for incorporating dynamic terrain features within DIS environments. We feel that these PDUs, the Destructible Entity PDUs tentatively approved for the DIS protocol version 3.0 [personal knowledge of the author], fully support the networking criteria just described.

A. DIS VERSION 3.0 DESTRUCTIBLE ENTITY PDUs

Six Destructible Entity PDUs were developed for the DIS protocol version 3.0. They include:

- Create/Modify PDU
- Deletion PDU
- Request ID PDU
- Reply ID PDU
- Request Object PDU
- Reply Object PDU

A destructible entity as it relates to these PDUs, or packets as they are sometimes called, is any static object whose state is capable of changing due to some force inflicted on it. Thus, not only are earthworks such as craters and berms included in this category, but also changes to objects such as trees or buildings are communicated as well with these packets. In general, these PDUs are used for recording and transmitting the state of the terrain (including the state of static objects on top of the terrain). They convey to the participating hosts in the simulation when a destructible entity has been created, modified, or deleted. In addition, they are used by a node to ensure that its terrain's appearance is consistent with the rest of the simulators' terrains. In order for a host to keep track of the state of the world, the host must maintain a list or table where information it has received

related to each player in regards to destructible entity creations, modifications, and deletions is recorded. A "terrain manager" may also be incorporated to help manage the terrain and these PDUs. A terrain manager's main purpose is to maintain the state of the common terrain, support queries from the "players," and update late joining participants. Thus, it is the terrain manager that communicates with the hosts via the Request and Reply PDUs mentioned above. In addition, the terrain manager may be implemented such that it handles issuing all Create/Modify or Deletion PDUs for all participants. Another option that may be employed is to allow the owning host, the host where the action occurred, to issue its own such PDUs. Overall, there are many ways to implement these packets from incorporating several terrain managers (one per geographic area) to relying solely on the participating nodes to maintain their own terrains and to keep each other informed and up-to-date. However, the method of implementation is not nearly as important as the concept that all DIS simulators know how to read, interpret, and send Destructible Entity PDUs most especially when two or more distinct DIS applications are interacting with each other. Therefore, the sections below describe how each of the Destructible Entity PDUs are to be used and outline the exact format of the fields of each of the PDUs. Following this listing an example of how we implemented these packets is presented.

1. Create/Modify PDU

This PDU is used to inform participants that another host created or modified a destructible entity. Thus, the Create/Modify PDU indicates to receiving nodes that they also must create this object or modify an entity (if it already exists) within their copy of the world based on the information contained within the PDU's fields. Additionally, the receiving hosts must update their destructible entity tables or lists (as previously described) to reflect receipt of this PDU. Descriptions of the Create/Modify PDU's fields are presented below with their exact system format following in Table 6.

a. ***PDU Header*** - The standard DIS Header.

b. ***Sequence Number*** - Incremented per owning host every time this PDU is sent regarding that host. This value is included within a structure that also contains the host and site IDs which are all attached to the outgoing PDU.

c. ***Object ID*** - Uniquely identifies the destructible entity.

d. ***Object Type*** - Identifies the name of the object - crater, building etc.

e. ***Object Material*** - Indicates what material to use to construct the entity.

f. ***Location*** - Identifies the location in world coordinates of the entity.

g. ***Orientation*** - Specifies the yaw, pitch, and roll of the object.

h. ***Height*** - Specifies the minimum and maximum height of the entity in body coordinates.

i. ***Length*** - Specifies the minimum and maximum length of the entity in body coordinates.

j. ***Width*** - Specifies the minimum and maximum width of the entity in body coordinates.

k. ***Appearance*** - Indicates via an enumerated type the specific appearance of the object such as flaming, smoking, or burnt.

2. Deletion PDU

This PDU is used to convey to participants that a host deleted a destructible entity. Hosts receiving the Deletion PDU are required to remove the object described from their worlds and to record receipt of this PDU within their destructible entity tables or lists. The fields listed below fully describe this PDU along with Table 7's listing of the fields' formats.

Field Size (bits)	Field Title	Description	
96	PDU Header	8-bit enumeration	Protocol Version
		8-bit unsigned integer	Exercise ID
		8-bit enumeration	PDU-Type
		8 bits unused	Padding
		32-bit unsigned integer	Time Stamp
		16-bit unsigned integer	Length
		16 bits unused	Padding
64	Sequence Number	16-bit unsigned integer	Site ID
		16-bit unsigned integer	Host ID
		16-bit unsigned integer	Sequence Number
		16 bits unused	Padding
16	Object ID	16-bit unsigned integer	
16	Object Type	16-bit unsigned integer	
16	Object Material	16-bit unsigned integer	
16	Padding	16-bits unused	
192	Entity Location	64-bit floating point	X
		64-bit floating point	Y
		64-bit floating point	Z
96	Entity Orientation	32-bit floating point	Psi
		32-bit floating point	Theta
		32-bit floating point	Phi
64	Height	32-bit floating point	Z-Minimum
		32-bit floating point	Z-Maximum
64	Length	32-bit floating point	X-Minimum
		32-bit floating point	X-Maximum
64	Width	32-bit floating point	Y-Minimum
		32-bit floating point	Y-Maximum
32	Appearance	32-bit enumeration	

Table 6: Create/Modify PDU

a. ***PDU Header*** - The standard DIS header.

b. ***Sequence Number*** - Incremented per owning host every time this PDU is sent regarding that host. This value is included within a structure that also contains the host and site IDs which are all attached to the outgoing PDU.

c. ***Object ID*** - Uniquely identifies the destructible entity.

d. ***Location*** - Identifies the location in world coordinates of the entity.

Field Size (bits)	Field Title	Description	
96	PDU Header	8-bit enumeration	Protocol Version
		8-bit unsigned integer	Exercise ID
		8-bit enumeration	PDU-Type
		8 bits unused	Padding
		32-bit unsigned integer	Time Stamp
		16-bit unsigned integer	Length
		16 bits unused	Padding
64	Sequence Number	16-bit unsigned integer	Site ID
		16-bit unsigned integer	Host ID
		16-bit unsigned integer	Sequence Number
		16-bits unused	Padding
16	Object ID	16-bit unsigned integer	
16	Padding	16 bits unused	
192	Entity Location	64-bit floating point	X
		64-bit floating point	Y
		64-bit floating point	Z

Table 7: Deletion PDU

3. Request ID PDU

The Request ID PDU is sent by a host in order to find out if any modifications to the terrain had taken place that it missed because it was late joining the simulation, it "dropped out" for a period of time, or hardware faults occurred that prevented the host from receiving terrain update information. The requesting node transmits a list of hosts' maximum sequence numbers in which each number indicates the last destructible entity information it has received in regards to these hosts. The node receiving this PDU responds with a similar list of maximum sequence numbers it has recorded per participant. The following list and Table 8 illustrate the fields used to communicate this data.

a. ***PDU Header*** - The standard DIS header.

b. ***Count*** - Specifies the number of hosts' maximum sequence numbers contained in the PDU.

c. ***Maximum Sequence Number List*** - Contains a list of all hosts and the maximum sequence number issued regarding each host that the requesting node has received.

Field Size (bits)	Field Title	Description	
96	PDU Header	8-bit enumeration 8-bit unsigned 8-bit enumeration 8 bits unused 32-bit unsigned integer 16-bit unsigned integer 16 bits unused	Protocol Version Exercise ID PDU-Type Padding Time Stamp Length Padding
16	Count	16-bit unsigned integer	
16	Padding	16 bits unused	
n*64 n= # of hosts	Maximum Sequence Number List	16-bit unsigned integer 16-bit unsigned integer 16-bit unsigned integer 16 bits unused	Site ID Host ID Sequence Number Padding

Table 8: Request ID PDU

4. Reply ID PDU

This PDU is used to respond to a host's transmission of a Request ID PDU. It contains a list of the last transmitted terrain state changes that occurred on each of the participating nodes within the simulation. The PDU indicates these latest changes by only including the highest sequence number sent regarding each owning host. A node receiving this information compares each host's sequence number in this PDU with the corresponding values in its destructible entity table and determines if there were any PDUs that it did not receive which now need to be requested. The fields employed in the Reply ID PDU are described below and in Table 9.

a. ***PDU Header*** - The standard DIS header.

b. ***Count*** - Specifies the number of hosts' maximum sequence numbers contained in the PDU.

c. ***Maximum Sequence Number List*** - Contains a list of all hosts and the maximum sequence number issued regarding each host that the responding node (in most cases the terrain manager) has received.

Field Size (bits)	Field Title	Description	
96	PDU Header	8-bit enumeration	Protocol Version
		8-bit unsigned integer	Exercise ID
		8-bit enumeration	PDU-Type
		8 bits unused	Padding
		32-bit unsigned integer	Time Stamp
		16-bit unsigned integer	Length
		16 bits unused	Padding
16	Count	16-bit unsigned integer	
16	Padding	16 bits unused	
n*64	Maximum Sequence	16-bit unsigned integer	Site ID
	Number List	16-bit unsigned integer	Host ID
		16-bit unsigned integer	Sequence Number
n= # of hosts		16 bits unused	Padding

Table 9: Reply ID PDU

5. Request Object PDU

This PDU is used by a participant to request Create/Modify or Deletion PDUs that it never received regarding an individual owning host - one Request PDU is sent per owning host's information needed. The requesting node indicates the PDUs desired by supplying bounds which when "Anded together" provide the condition that all retransmitted PDUs must match. Thus, the responding node replies with packets that satisfy an "And" of **all** of the fields outlined below (except a.). The system format of these fields is presented in Table 10.

a. ***PDU Header*** - The standard DIS header.

b. ***Minimum Sequence Number*** - Specifies the earliest transmitted PDU needed regarding a specific host.

c. ***Maximum Sequence Number*** - Specifies the latest transmitted PDU needed regarding a specific host; thereby requesting all PDU's previously transmitted regarding one host from the minimum sequence number to the maximum sequence number inclusive.

d. ***Minimum Time*** - Specifies the oldest PDU (in terms of time) regarding a specific host that the requesting host wants.

e. ***Maximum Time*** - Specifies the youngest PDU (in terms of time) regarding a specific host that the requesting host wants; thereby requesting all PDU's previously transmitted regarding one host from the minimum time to the maximum time inclusive.

f. ***Minimum Geographic Bounds*** - Specifies the minimum bounds in world coordinates within which a retransmitted PDU's destructible entity must lie.

g. ***Maximum Geographic Bounds*** - Specifies the maximum bounds in world coordinates within which a retransmitted PDU's destructible entity must lie; thereby requesting all PDU's previously sent regarding one host from the minimum bounds to the maximum bounds inclusive.

6. Reply Object PDU

The Reply Object PDU is used to send a list of previously transmitted PDUs regarding a specific host in response to the Request Object PDU. The PDUs contained within this packet match an "And" of the bounding fields within the Request Object PDU. Upon receipt of a Reply Object PDU the requesting host updates its world's state and its table used for recording the receipt of Destructible Entity PDUs.

Field Size (bits)	Field Title	Description	
96	PDU Header	8-bit enumeration	Protocol Version
		8-bit unsigned integer	Exercise ID
		8-bit enumeration	PDU-Type
		8 bits unused	Padding
		32-bit unsigned integer	Time Stamp
		16-bit unsigned integer	Length
		16 bits unused	Padding
64	Minimum Sequence Number	16-bit unsigned integer	Site ID
		16-bit unsigned integer	Host ID
		16-bit unsigned integer	Sequence Number
		16 bits unused	Padding
64	Maximum Sequence Number	16-bit unsigned integer	Site ID
		16-bit unsigned integer	Host ID
		16-bit unsigned integer	Sequence Number
		16 bits unused	Padding
32	Minimum Time	32-bit unsigned integer	
32	Maximum Time	32-bit unsigned integer	
192	Minimum Geo-graphic Bounds	64-bit floating point	X
		64-bit floating point	Y
		64-bit floating point	Z
192	Maximum Geo-graphic Bounds	64-bit floating point	X
		64-bit floating point	Y
		64-bit floating point	Z

Table 10: Request Object PDU

a. ***PDU Header*** - The standard DIS header.

b. ***Count*** - Specifies the number of PDUs sent (Create/Modify or Deletion) regarding one host's destructible entities.

c. ***Create/Modify Parameters*** - The following fields are part of this structure and are a duplication of the Create/Modify PDU's fields (which also encompass the fields of the Deletion PDU). The fields are described in the Create/Modify PDU section. In addition, a variable number, n, of these structures may be attached to the Reply Object PDU depending on how many packets related to a particular host need to be retransmitted.

(1) Sequence Number.

(2) Object ID.

(3) Object Type.

(4) Object Material.

(5) Location.

(6) Orientation.

(7) Height.

(8) Length.

(9) Width.

(10) Appearance.

B. DESTRUCTIBLE ENTITY PDU IMPLEMENTATION

Our implementation of the above PDUs occurs within a previously existing 2D grid modeler which periodically launches a missile, models its flight, determines when and where it hits the ground or an object, and consequently sends a DIS Detonation PDU [IST93] to the network. The modeler's displayed grid represents the X and Y world coordinates of a terrain database. In addition, besides transmitting DIS PDUs, the modeler also receives and processes DIS packets. One such packet is the Entity State PDU which is used to communicate the existence and state of all entities other than destructible entities [IST93]. By having the ability to process these packets, the modeler is able to not only represent the presence of its host's objects including the tracking of the missile's flight, but also other nodes' entities by placing icons at the X and Y coordinates of the grid according to where they actually exist within the world.

The modeler's ability to communicate with the network and display existing objects in this way, made it a logical place to implement Destructible Entity PDUs and verify proper communications between a terrain manager and participating hosts. Both the terrain manager and the "players" operate under a copy of this modeler which has been adjusted to communicate with a file of destructible entity processing functions. Basically, the players' modelers operate as just described with the added feature of having the ability to receive, process, and transmit Destructible Entity PDUs. However, the participants in the

Field Size (bits)	Field Title	Description	
96	PDU Header	8-bit enumeration	Protocol Version
		8-bit unsigned integer	Exercise ID
		8-bit enumeration	PDU-Type
		8 bits unused	Padding
		32-bit unsigned integer	Time Stamp
		16-bit unsigned integer	Length
		16 bits unused	Padding
16	Padding	16-bits unused	
16	Count	16-bit unsigned integer	
n*640	Create/Modify Parameters	Sequence Number:	
		16-bit unsigned integer	Site ID
		16-bit unsigned integer	Host ID
		16-bit unsigned integer	Sequence Number
(n = number of retransmitted PDUs)		16 bits unused	Padding
		16-bit unsigned integer	Object ID
		16-bit unsigned integer	Object Type
		16-bit unsigned integer	Object Material
		16 bits unused	Padding
		Entity Location:	
		64-bit floating point	X
		64-bit floating point	Y
		64-bit floating point	Z
		Entity Orientation:	
		32-bit floating point	Psi
		32-bit floating point	Theta
		32-bit floating point	Phi
		Height:	
		32-bit floating point	Z-Minimum
		32-bit floating point	Z-Maximum
		Length:	
		32-bit floating point	X-Minimum
		32-bit floating point	X-Maximum
		Width:	
		32-bit floating point	Y-Minimum
		32-bit floating point	Y-Maximum
		32-bit enumeration	Appearance

Table 11: Reply Object PDU

simulation are not permitted to send Create/Modify, Deletion, Reply ID, or Reply Object PDUs. Only the Terrain Manager responds to hosts' requests with the two types of reply PDUs and transmits Create/Modify and Deletion PDUs. The terrain manager on the other hand, does not participate as a player - it does not shoot missiles nor does it request periodic updates of destructible entities. However, both the players and the terrain manager display the same state of the world on their 2D grids.

The general differences just described between the terrain manager and the participating hosts are handled by calls to the destructible entity functions file. Therefore, the specific features of this file and how such simulators interact with the file's functions are described in the next section with a more precise description of the terrain manager's and hosts' interaction with these functions following.

1. Destructible Entity Functions File

The destructible entity functions file not only contains functions that support the use of Destructible Entity PDUs, but it also provides a matrix structure, a doubly linked list, for keeping track of all destructible entities owned by each host. The matrix is composed of a linking of all the hosts participating in the simulation and from each of these hosts stems a list of the destructible entities for which the host is responsible. Each of the destructible entity nodes in the hosts' lists contain the fields of the Create/Modify PDU with the exception of the PDU Header. The set of functions within this file are primarily used for sending Destructible Entity PDUs and for comparing and updating the host matrix. Figure 14 displays the structures employed and Figure 15 contains all of the functions' prototypes. In addition, Figure 16 visualizes the host matrix structure, TotalObjsNode, defined in Figure 14. And finally, the sections below further describe the routines defined in Figure 15 and the file's overall handling of the host matrix.

a. mallocTotalObjsNode()

This function allocates space for a host in the matrix and returns a pointer to that area of memory. It is used by a node when it first enters the simulation in order to build

```
//Header file for all destructible entity related information.
//*********************************Structures*****************************
//*****Structure for applications to use to supply data to be transmitted in a
//*****Create/Modify PDU.
typedef struct {
    SequenceNum             dest_ent_seq_num;
    ObjectID                object_id;
    ObjectType              dest_ent_object_type;
    ObjectMat               dest_ent_object_mat;
    short                   padding16;
    EntityLocation          dest_ent_location;
    EntityOrientation       dest_ent_orientation;
    HeightZ                 dest_ent_height;
    LengthX                 dest_ent_length;
    WidthY                  dest_ent_width;
    Appearance              dest_ent_appearance;
} C_M_DestEnt;


//*****Structure for applications to use to supply data to be transmitted in a
//****Deletion PDU.
typedef struct {
    SequenceNum             dest_ent_seq_num;
    ObjectID                object_id;
    EntityLocation          dest_ent_location;
} Del_DestEnt;


//*****Structure for matrix that contains all hosts and their created,
//*****modified, or deleted objects. CreateModDelObjNode is a structure
//*****that resides in the pdu.h file in conjunction with the Reply Object PDU.
//*****The matrix is global to the entire source code file.

typedef struct {
    CreateModDelObjNode     *object;
    struct total_objs_node  *nexthost;
} TotalObjsNode;

//*************************end Structures*****************************
```

Figure 14: Structures that Support the Use of Destructible Entity PDUs

its host matrix. The function is also employed by the node to add additional late joining hosts to its matrix.

```
//Header file for all destructible entity related information (cont.).
//*****************************Prototypes*******************************
//*****Allocates space for matrix.
TotalObjsNode *mallocTotalObjsNode();
//*****Reads in all hosts from file and builds empty matrix of hosts.
int readhostfile();
//*****Provides a copy of the destructible entity matrix.
TotalObjsNode * get_DestEnt();
//*****Creates Create/Modify PDU from data sent via the parameter, transmits
//*****the PDU, and updates matrix with the information sent.
void sendCM (C_M_DestEnt);
//*****Creates Deletion PDU from data sent via the parameter, transmits
//*****the PDU, and updates matrix with the information sent.
void sendDel (Del_DestEnt);
//*****Sends a Request ID PDU which includes all hosts in current matrix.
void sendReqID ();
//*****Sends a Request Object PDU.
void sendReqObj(int);
//*****Sends a Reply ID PDU which includes all hosts in current matrix.
void sendReplyID();
//*****Sends a Reply Object PDU.
void sendReplyObj(RequestObjDestEntPDU *);
//*****Adds Create/Modify PDU information to matrix.
void updateCM(CreateModDestEntPDU *pdu);
//*****Adds Delete PDU information to matrix.
void updateDel(DeleteDestEntPDU *pdu);
//*****Compares information in Reply ID PDU and sends a Request Object PDU.
void compareReplyID(ReplyIDDestEntPDU *pdu);
//*****Updates matrix with previously sent PDUs contained in Reply Object PDU.
void updateTerrain(ReplyObjDestEntPDU *pdu);
//*****Prints all fields in matrix.
void printMatrix ();
//*****Routines below print all fields of PDU parameters.
void printCM (CreateModDestEntPDU *pdu);
void printDel (DeleteDestEntPDU *pdu);
void printReqID(RequestIDDestEntPDU *pdu);
void printReqobj (RequestObjDestEntPDU *pdu);
void printRepID (ReplyIDDestEntPDU *pdu);
void printRepObj (ReplyObjDestEntPDU *pdu);
//*************************End Prototypes*******************************
```

Figure 15: Functions that Support the Use of Destructible Entity PDUs

b. readhostfile()

Readhostfile reads in the list of host and site IDs from a file and builds the host matrix with this data (calls mallocTotalObjsNode to create list nodes for each host). The function returns a one (true) if it was successfully able to read the file.

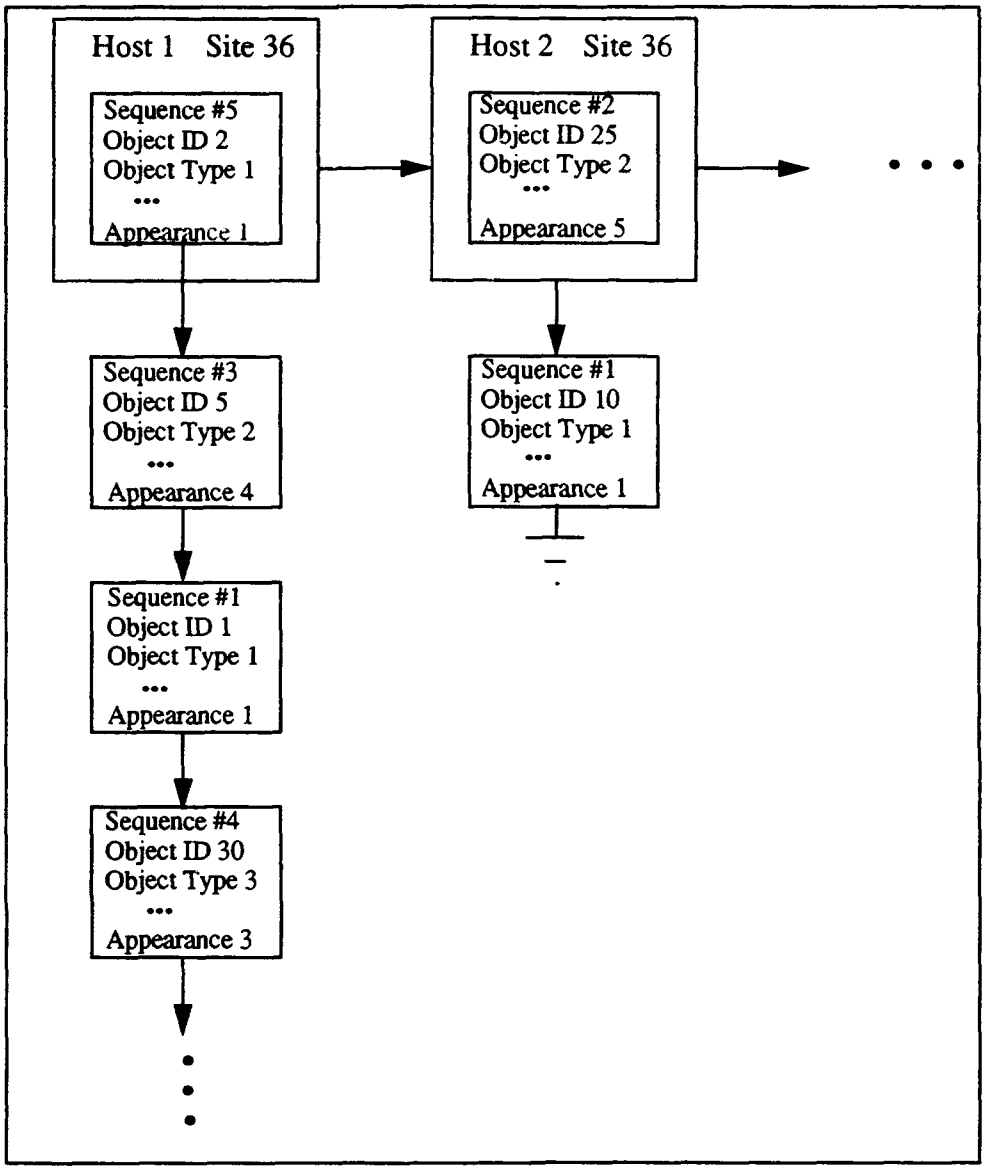

Figure 16: Example Host Matrix

c. get_DestEnt()

This function simply returns a copy of the entire host matrix. Get_DestEnt is useful to a system's draw process in that this process can obtain and scan its own copy of the host matrix and determine what destructible entities need to be displayed.

d. sendCM()

In general, sendCM receives a parameter of type C_M_DestEnt (illustrated in Figure 14) and sends a Create/Modify PDU based on the fields of that argument. However, prior to transmitting the PDU, this function determines if the object at hand (parameter) is new and if so, it assigns the entity a unique object ID. The function knows that the object is new if the parameter's object ID field equals zero. After this assignment, sendCM searches the host matrix for the owning host's list of objects. Since the destructible entity file of functions maintains the host matrix such that the first object in a host's list is the one with the highest sequence number for that host, this algorithm assigns the PDU a sequence number one higher than the one at the head of its owning host's list. At this point, the function searches the proper host's list for an entity with the same object ID as the parameter's. If a match is found (parameter is a modification) the matching list node is overwritten with the new data and placed at the head of the host's list since it now contains the highest sequence number. If on the other hand, the object is new, then a new node is created, filled with the new information, and inserted at the head of the host's list. Furthermore, if the first node's padding field's value is -1 then that node is simply overwritten with the parameter's values since this padding value signifies a host place holder (host does not own any destructible entities as of yet) ensuring that the padding value is changed to zero. Finally, after the matrix has been updated, the Create/Modify PDU is sent.

e. sendDel()

This function operates identically to sendCM except that it handles the deleted destructible entities. An object ID is not assigned since a deletion indicates that the object already exists. However, the sequence number assignment and insertion of the entity into the host matrix are performed in the same way as they are in sendCM with one important exception. When the object is inserted in the proper position in the matrix, the object's appearance field is assigned zero to signify that the entity has been deleted. Finally,

just as sendCM sends a Create/Modify PDU, this function puts a Deletion PDU out on to the network.

f. sendReqID()

SendReqID sends a Request ID PDU which includes all of the hosts and their highest sequence numbers contained in the host matrix. This function simply traverses the host list and copies the sequence number from the head of each individual host's lists (since the highest sequence number is always maintained at the head of these lists) into the PDU. In addition, the count field of the Request ID is filled with the number of hosts contained within the PDU.

g. sendReplyID()

This function sends a response to a Request ID PDU in the form of a Reply ID PDU. Since the format of this packet is identical to that of the Request ID PDU, sendReplyID fills this response PDU in the same way that sendReqID fills its corresponding PDU.

h. sendReqObj()

SendReqObj fills in the boundary fields of a Request Object PDU and sends this packet in order to request previously transmitted Destructible Entity PDUs regarding one specific host. This function must be called with two parameters. The first is the hosts's highest sequence numbered node contained in the calling player's host matrix and the second is the maximum sequence number received by the player via a Reply ID PDU for that host. Thus, these two arguments are used by sendReqObj to fill the minimum and maximum sequence number fields of the Request Object PDU. This function also takes into account the possibility that the host is new to the requesting player and thus, if this is the case, the minimum sequence number field is assigned the number one in an effort to request all Destructible Entity PDUs sent regarding this host. The remaining boundary fields of the Request Object PDU are presently just assigned hard coded values so that requests for

objects are only limited to the sequence number boundaries. The other limiting fields need to be properly incorporated in future improvements to this code perhaps by incorporating the other fields included in the first parameter used to call this function.

i. sendReplyObj()

SendReplyObj searches for previously transmitted PDUs recorded in the responding host's matrix that meet the condition fields of the Request Object PDU parameter used to call this function. The search initially involves finding the owning host's list. Once located, the list is then traversed testing each node against the boundaries set in the Request Object PDU. Any node that satisfies an "And" of all of these boundaries is copied and attached to a Reply Object PDU, and upon completion of the traversal of the list the PDU is transmitted.

j. updateCM()

UpdateCM takes in a Create/Modify PDU and updates the host matrix. The update procedure is the same as described in the sendCM section with the exception that the PDU received already contains the proper object ID and sequence number, meaning that the assignment of these fields that takes place in sendCM is not performed in this function. However, this algorithm does not assume that the incoming PDU contains the highest sequence number transmitted regarding the owning host. If the number is not the highest then the owning host's list is updated in one of two ways depending on whether the data within the PDU refers to a new object or to a modification of an already existing entity. If the object is new then the PDU's information is placed in a new node added to the tail of the list. On the other hand, if the PDU contains modification data, then the object's node in the list is just overwritten with this new information as long as the new sequence number is indeed higher than this node's. But, if the PDU's sequence number is the highest of those existing in the owning host's list, then its information is placed at the head of this list as presented in the section describing sendCM.

k. updateDel()

This function updates the host matrix with information contained within a Deletion PDU. The update procedure is exactly the same as the one implemented in updateCM. One additional assignment, however, is implemented when the PDU's information is copied into the proper node. This assignment involves replacing the appearance field value of the node with zero to indicate that the object has been deleted since the Deletion PDU does not contain this field for copying. In addition, it should be mentioned that even if the deleted object does not exist in the host matrix (for whatever reason), the data is copied into the list (in the same way that a Create/Modify PDU's data is). The reason that this deleted object is placed in the list is so that a host can decipher what information regarding this entity should be displayed in case PDUs related to it arrive out of sequence. (e.g. If the Deletion PDU's sequence number is five, but a Create/Modify PDU for the same object with a sequence number of four arrives after the Deletion PDU, the receiving host would know not to display the object.)

l. compareReplyID()

CompareReplyID accepts a Reply ID PDU and determines if the host matrix is missing destructible entity information regarding any of the hosts involved in the simulation. The function compares every participating node's maximum sequence number contained within the PDU with the corresponding maximum sequence number in the host matrix (the sequence number at the head of each host's list). If during each host's comparison the sequence number in the PDU is greater than the one for the same host in the host matrix then a Request Object PDU is sent using the sendReqObj function previously described. If a node in the PDU does not exist in the host matrix then a new node place holder is created for that host in the matrix and sendReqObj is called in an effort to obtain all Destructible Entity PDUs sent regarding the new host.

m. updateTerrain()

This function receives a Reply Object PDU containing retransmitted PDUs and updates the host matrix with each of these PDUs in the same manner that updateCM does.

n. printMatrix() and All Print Routines

All of the print routines contained within the destructible entity functions file are provided for debugging purposes. PrintMatrix displays all of the destructible entities information (essentially all of the fields that compose a Create/Modify PDU) in host ID numerical order from head to tail. In between the start of each host and the start of the one immediately following are the entities belonging to the first host displayed from head to tail. For example, if printMatrix were to print the list as it appears in Figure 16 it would display host 1 with the data of Object IDs 2, 3, 5, and 30 following in that order and then host 2 would be displayed with the data of Object ID's 25 and 10 following. The remaining print functions, printCM, printDel, printReqID, printReqObj, printRepID, and printRepObj, display all of the fields of their respective PDU arguments.

2. Player and Terrain Manager Interaction

Our implementation of a terrain manager interacting with participating hosts using the 2D modeler previously described begins by starting the terrain manager's copy of the modeler. The terrain manager must be started first to ensure that it witnesses all of the simulation's world events. Hosts may be brought on line at any time thereafter. To prepare for destructible entity processing, both the terrain manager and participating hosts initially call readhostfile to read in the host and site IDs in order to set up the host matrix. In addition, when a host first enters the simulation, it immediately sends a Request ID PDU by calling sendReqID to determine if it has missed any transmitted Destructible Entity PDUs. Upon receiving a Reply ID PDU from the terrain manager, the host calls compareReplyID which evaluates the PDU and consequently sends a Request Object PDU by calling sendReqObj if it determines that the host is indeed missing destructible entity

data. Again, the terrain manager responds with a Reply Object PDU via a call to sendReplyObj, and finally, the host updates its host matrix by calling updateTerrain. A host not only calls sendReqID when it first enters the simulation, but also periodically during the simulation - approximately every minute - to ensure that the state of its terrain is always consistent with the terrain manager's world. Therefore, this full sequence of events which is graphically displayed in Figure 17 may occur many times during the simulation.

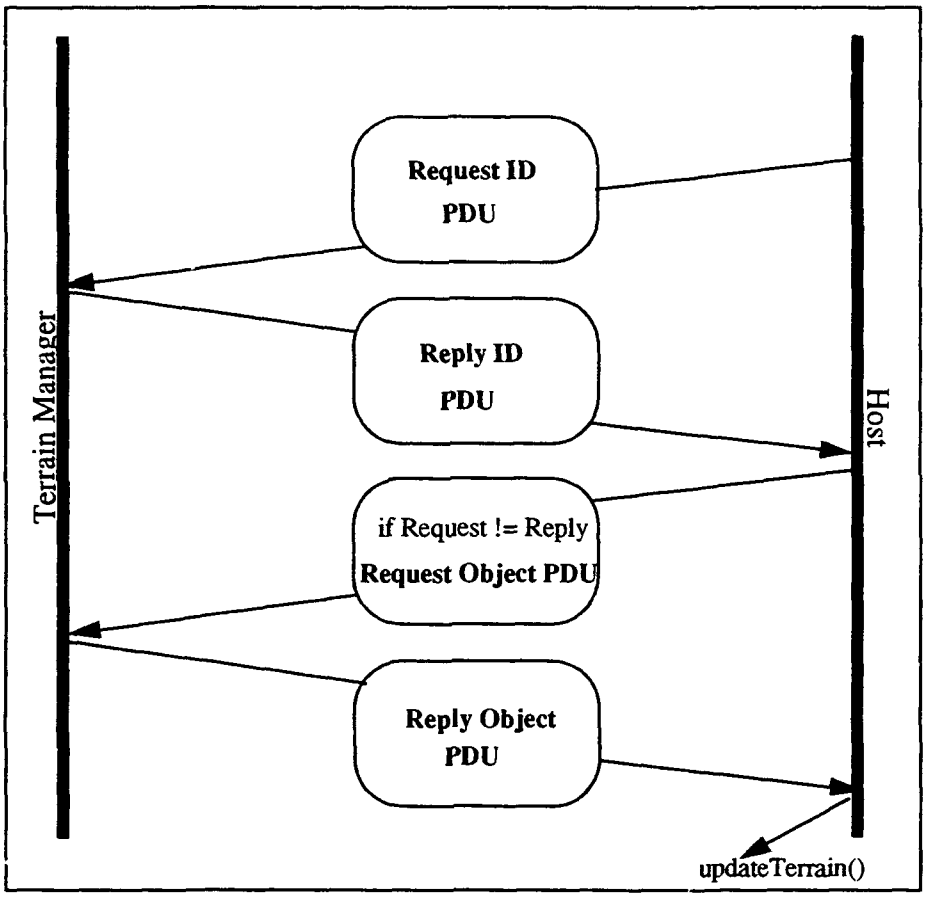

Figure 17: Destructible Entity Request and Reply PDU Sequence

After a host is initialized, it begins to periodically shoot missiles one at a time. When a missile blows up, the owning host sends a Detonation PDU. Only the terrain manager in this implementation reads and interprets Detonation PDUs. Therefore, when the terrain manager receives this packet, it examines its result field to determine if the missile

hit the ground or came close enough to cause damage to the terrain. If one of these two results occurred, then the terrain manager sends a Create/Modify PDU by calling sendCM with a parameter of type C_M_DestEnt (as defined in Figure 14) that it has filled. Part of the parameter's field values are copied from the Detonation PDU. These include the location of the detonation and the site and host IDs of the player that sent the PDU. The rest of the variable's values, with the exception of the object ID and sequence number which are uniquely assigned in sendCM, are simply hard coded and indicate a default crater was formed. This terrain manager and host interaction is illustrated in Figure 18.

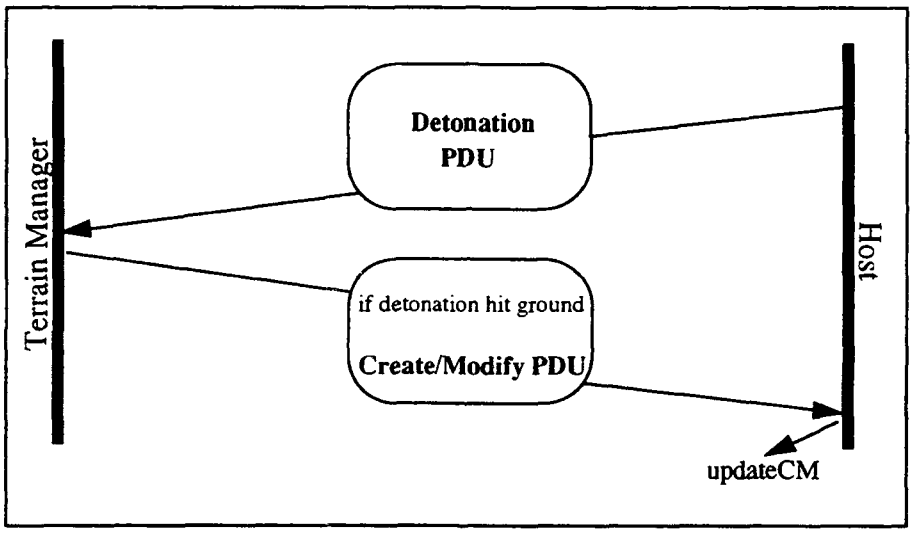

Figure 18: Creation or Modification of a Destructible Entity

In order to eliminate the use of constantly identical hard coded values, the Detonation PDUs' fields describing the munitions used would have to be evaluated to accurately determine the effects that the detonations had on the terrain or other static objects. After such determinations are made, the parameter's fields which ultimately become the Create/Modify PDU fields would be filled with data that describes these effects well enough that the receiver of the PDU would be capable of displaying the same detonation result. This type of evaluation needs to be incorporated, for it will definitely

make the terrain appear more realistic in future applications. (For a complete description of the Detonation PDU see [IST93].)

The final way in which the terrain manager communicates world changes to the participants of the simulation is by sending a Deletion PDU for an object that already exists. This PDU transmission was implemented in the form of a key punch in order to ensure proper handling of this packet. When a user operating the terrain manager pushes the "D" key, the terrain manager calls sendDel with a variable of the type Del_DestEnt (as defined in Figure 14) which updates its host matrix to reflect that a deletion has occurred (sets the object's appearance field value to zero) and transmits the Deletion PDU. Hosts on the network accept this PDU and call updateDel to process it.

Overall, the terrain manager and hosts are able to interact correctly using all six Destructible Entity PDUs. This proper acceptance and processing of the PDUs is verified by printing host matrices of simulation participants (including the terrain manager) using the print functions provided by the destructible entity functions file. In addition, the modeler's grid also verifies this correct interaction. During each cycle through the main simulation loop of the program a call is made to a draw process. This process draws the grid and places any existing destructible entities (using icons) at locations on the grid that correspond with the entities' positions on the terrain database. In order to know which entities' it should represent, the draw procedure calls get_DestEnt to obtain its own copy of the host matrix. It then searches the entire matrix and displays existing entities using the location fields within the entities' nodes. But, if the draw process finds an entity whose appearance value is zero, it does not display that entity since it has been deleted. Therefore, it was by way of both of these displays that we were able to confirm that our implementation transmitted, accepted, and processed the Destructible Entity PDUs correctly. Furthermore, to support our belief that these PDUs are capable of being implemented within a real-time simulator we conducted a series of performance tests with them as described in the next section.

3. Destructible Entity PDU Performance

We tested the transmission of Destructible Entity PDUs in terms of how they affect Ethernet capacity. The first test involved two hosts continuously sending Create/Modify PDUs at a rate of 30 packets per second meaning that a total of 60 Create/Modify PDUs were traversing the Ethernet at any one time. This network load used only 1.5% of the total Ethernet capacity. Next we added a third host to this simulation that continuously sent Request ID PDUs every ten seconds to which the other two hosts responded by transmitting Reply ID PDUs. In addition, if the new host determined from the receipt of Reply ID PDUs that it was missing data regarding destructible entities, it sent a Request Object PDU to which the initial two hosts responded with Reply Object PDUs. This particular circumstance (object request and replies) only occurred when the third host first joined the simulation. Even though Request ID and Reply ID PDUs were traversing the network every ten seconds in addition to the 60 Create/Modify PDUs that were being sent to the network every second, the effect on the Ethernet capacity did not change from that of the first test.

Our next two tests involved sending 120 Create/Modify packets per second on to the Ethernet. Two hosts were used with each sending 60 of these PDUs per second. With just these packets on the network only 2.2% of the Ethernet capacity was used. Again, we added a third host that interacted with the first two hosts exactly as described in the previous paragraph's second test. As expected, the amount of Ethernet capacity used remained constant with this addition as it had in the comparison of the first two tests above.

The results of our tests indicate Destructible Entity PDUs are capable of being incorporated within a real-time distributed simulator without hampering the simulator's performance. This statement is supported by the fact that our hosts sent PDUs on to the network at optimal real-time rates and only used a very tiny portion of the Ethernet's capacity. Furthermore, the PDU transmission rates we incorporated are higher than what would occur in many distributed simulator scenarios meaning that even less of the network's capacity would end up being used.

C. DESTRUCTIBLE ENTITY PDU SUMMARY

The Destructible Entity PDUs are indeed a viable solution to improving the realism of the terrain within DIS simulators. Our performance results indicate that these PDUs can be handled by a real-time distributed system without hampering the overall communications process. In addition, our implementation of these packets demonstrates how destructible entity information can be incorporated relatively easily within a distributed system. However, there are other conditions that must be taken into account to properly incorporate these PDUs. Examples include the one provided earlier in this document regarding examining Detonation PDU munitions' fields to determine just how much damage a detonation caused or when a Deletion PDU should be transmitted. These issues and others need to be addressed and implemented. Although, we have yet to incorporate such events, we have developed a testbed which provides both written and graphical verification of destructible entity communications. Thus, this testbed can be further used to test other such issues or events relating to destructible entities.

VI. EXPLOSIONS

During times of war, weapons are continually being fired and are consequently impacting the terrain, objects, and humans. Objects when struck by a significant force burst into many fragments. However, many real-time virtual battlefield simulators only represent such a stricken entity with an animated flash followed by a "dead model" representation. Examples of dead models include a blackened building, a bent over tree, or a tank with a dipped turret. But, only modeling a "killed" object by immediately switching the "alive" entity with a "dead" one detracts from the realism of the synthetic battlefield.

As stated previously, explosive effects have been created for NPSNET, but they run in versions preceding the incorporation of Performer. Taking advantage of Performer's ability to optimize a simulation's performance on Silicon Graphics systems means that all geometry within the simulation must be made compatible with this toolkit's low level geometry (pfGeoSets). Performer provides routines for converting several common geometric formats to that of Performer's. Therefore, all objects read into NPSNET are converted to Performer's geometric format and thus, it was our goal to design an algorithm that has the ability to explode any object that Performer is capable of rendering.

A. INTRODUCTION TO THE EXPLOSION FUNCTION

In general, we wrote a function that accepts as its parameter a pointer to a pfGeoSet and retrieves the necessary information about the pfGeoSet in order to tear it apart into its polygon primitives. These individual primitives are then used to create a blowing up effect due to some impact the object sustained. However, often times the polygons are not readily accessible and therefore have to be converted to an "explodable format -" a task this function also performs. An example of where this transformation is needed is when sharing of vertices' memory locations among primitives is incorporated. Since the explosion occurs by calling this function multiple times, the pfGeoSet is converted only once so that it is permanently changed. The motion of the polygons is physically-based using equations that incorporate force direction, rotation of the object about the axis normal to its force

direction, and ballistic motion, all of which will be explained later in this chapter. As just mentioned, an object fully blows up by calling this function repeatedly until the explosive motion has ceased - this function ends all such motion after it has been called 150 times.

1. Function's Parameters

Since the explosion routine has to be continually called for the same pfGeoSet, the function's parameters are very important. In other words, users incorporating the function must take care to ensure that the parameters passed in for a particular pfGeoSet are those that belong to this object, especially if many explosions are occurring at the same time. The function's prototype is defined as follows (variable types are in italics and pfVec3 is a structure holding three float values):

blowup (*float* force, *int* &call_num, *pfGeoSet* *gset1, *pfVec3* *Vel,
double deltatime, *pfVec3* where);

The arguments force, Vel, and where are all needed for the motion equations. Force is the initial force inflicted on the object at the point of impact. Vel is an array holding the previous call's calculated velocities specified in X, Y, and Z components for each of the primitives of the pfGeoSet (except for in the first call where the initial velocity is computed). The velocity array is important, because the function employs ballistic motion equations which need the previous call's velocities for each of the primitives of a pfGeoSet in order to compute the new locations of each polygon. Thus, the array passed in must be large enough to hold velocity information for all primitives of a pfGeoSet. The last of the motion equations' parameters, where, contains the location of the weapon's impact on the pfGeoSet. The rest of the arguments include a pointer to the pfGeoSet that has been struck, gset1; a time variable which contains the time it took the calling program to cycle through its last simulation loop, deltatime; and a variable that specifies the number of times the function has been called for the pfGeoSet pointer parameter, call_num. Call_num is particularly important for two reasons. The first reason is that the motion equations require that the instantaneous velocity be calculated **only** the first time the function is called per pfGeoSet. Thus, this argument indicates when this particular call has occurred. The other

purpose of call_num is to inform the explosion routine when the explosion has completed - after 150 calls.

2. Performer pfGeoSet Functions

Although the function is provided with the above outlined parameters, it also needs to have access to information related to the construction of the pfGeoSet parameter particularly in the explodable format conversion part of the function. Performer provides a set of "get" routines related to pfGeoSets which supply this data. Specifically, they provide information regarding the object's positions of vertices, normals' values and binding (per primitive, per vertex, or overall), texture coordinates and binding, and color's RGBA values and binding. In addition, if the values of these attributes which are stored in an array are indexed, then the indices which are also stored in an array may be obtained as well. Examples of calls for such information are as follows:

```
pfGetGSetAttrLists(gset1, PFGS_NORMAL3, (void**)&sn2, &sni2);
col_bind = pfGetGSetAttrBind(gset1, PFGS_COLOR4);
```

In the first example, pfGetGSetAttrLists returns the array of values for the normals of the pfGeoSet gset1 in the variable sn2 and the array of indices to the normals' values if it exists (otherwise NULL is returned), in the variable sni2. This same function is used to obtain the values (and indices) of the vertices, colors, and texture mapping coordinates. The arrays returned for each of these attributes are composed of Performer's type pfVec3 or pfVec4 structures. PfVec3 is used for vertices, texture coordinates, and normals since it is a structure holding floating point values for the X, Y, and Z components of these attributes. PfVec4 is used for colors since it too holds floating point values, but for the RGBA components of the colors. Thus, every position in an array holds either a three or four component structure. In the second example, pfGetGSetAttrBind returns gset1's color's type of binding in the variable col_bind. Furthermore, additional "get" functions exist for obtaining the type of primitive (points, lines, linestrips, triangles, tristrips, and quads) that composes a pfGeoSet as well as the amount of primitives included in the object. In addition,

a function is also provided for obtaining the amount of vertices that are contained in each tristrip or linestrip primitive that make up a pfGeoSet, since a pfGeoSet may contain multiple strip primitives of differing sizes. There are other "get" functions, but they are not applicable to the explosion routine.

The compliment to the "get" routines are the "set" functions. These are important, because once the conversion algorithm deciphers the construction properties of the pfGeoSet, it performs necessary conversion calculations and then uses the set functions to redefine the pfGeoSet permanently - completing the conversion to explodable format. The "set" routines allow a program to reassign all of the parameters mentioned in the preceding paragraph. However, the arrays are not modified as easily as the other pfGeoSet attributes. When the explosion function's conversion algorithm needs to expand an array in order to hold more data (e.g., to eliminate indexing - a concept explained in the next section), a new array of the same type must be created with enough memory allocated to hold all of the values. This array is then assigned these values and the pfGeoSet is consequently "reset" with this new array.

The concepts regarding Performer's "get" and "set" functions presented in this section are all vital to the operation of the explosion function. They are of particular importance to the explodable format conversion algorithm within this file. While a general description of these functions has been furnished in this section, their specific use is outlined in Section B.

B. EXPLOSION FUNCTION DETAILS

The explosion function is comprised of two major algorithms. The first sequence of commands involves obtaining information regarding the pfGeoSet parameter and then converting it, if necessary, to explodable format. The latter part of the function computes the movement of the polygons since the last call to the function using physically-based concepts that include ballistic motion, force direction, and object rotation about an axis that is normal its force vector.

1. Explodable Format Conversion Algorithm

a. Elimination of Indexing

When the explosion function is first called, it gathers information about the pfGeoSet it has been passed using the Performer functions described above. Next, it determines if the vertices array is indexed. If this indexing exists then by Performer's pfGeoSet definition rules, all other attribute's arrays are indexed as well. Indexing is mainly used to save memory in that it allows primitives that have common vertices with other primitives of the same pfGeoSet to share the same memory locations for each of those vertices. However, in order to break up the pfGeoSet into its polygons and make them move independently to create the explosive effect, this indexing must be eliminated which is the next task the algorithm performs. This removal of indexing first entails creating a new array of values for each of the pfGeoSet's attributes where each array is large enough to hold the values dictated by its corresponding attribute's binding type (also equal to the number of indices in the index array for the attribute at hand). For example, if the normals are bound per primitive then the normals values' array is defined to be large enough to hold a separate normal value for each primitive. The next step to eliminating indexing is to assign each of these arrays values from their original corresponding arrays indexed by the their lists of matching indices. Finally, the pfGeoSet is "reset" using the function pfGeoSets which accepts the same parameters as its counterpart, pfGetGSetAttrLists, including an additional binding argument. Thus, modification to the pfGeoSet is performed by calling this function for each of the four attributes, vertices, normals, textures, and colors, and included in each call is the new expanded array of values for the corresponding attribute and the value of NULL as a filler for the index array argument. Figure 19 presents an example of obtaining general information regarding a pfGeoSet, how this information is used, and how indexing of the vertices' positions array is eliminated.

```
//********** Find out about pfGeoSet.**********
prim_type = pfGetGSetPrimType (gset1);
num_prims = pfGetGSetNumPrims(gset1);
//If primitives are constructed of tristrips then retrieve the array that contains
//the number of vertices per primitive.
 if (prim_type == PFGS_TRISTRIPS)
   {lengths = pfGetGSetPrimLengths(gset1); }
//****Get attributes' bindings.
norm_bind = pfGetGSetAttrBind(gset1, PFGS_NORMAL3)
col_bind = pfGetGSetAttrBind(gset1, PFGS_COLOR4);
//Really don't need next line - only one way to assign vertices.
verts_bind = pfGetGSetAttrBind(gset1, PFGS_COORD3);
text_bind = pfGetGSetAttrBind(gset1, PFGS_TEXCOORD2);
//****Get attribute arrays and indices arrays if present.
pfGetGSetAttrLists (gset1, PFGS_COORD3,(void **) &sv2, &svi2);
pfGetGSetAttrLists (gset1, PFGS_NORMAL3, (void **) &sn2, &sni2);
pfGetGSetAttrLists (gset1, PFGS_TEXCOORD2, (void **) &st2, &sti2);
pfGetGSetAttrLists (gset1, PFGS_COLOR4, (void **) &sc2, &sci2);
//******* Compute some helpful info - number of vertices per primitive.*******
if (prim_type == PFGS_QUADS)
   {num_verts = sides4 * num_prims;}
if (prim_type == PFGS_TRIS)
   {num_verts = sides3 * num_prims;}
if (prim_type == PFGS_TRISTRIPS)
   {
     for (int h = 0; h<num_prims; h++)
        {num_verts += lengths[h];}
   }
//This "if" encompasses all attributes' elimination of indices since when
//one attribute is indexed all are.
if (svi2 != NULL)
{
   //********** Handle Vertices indexing.*********
   //Allocate memory so that all primitives' vertices have separate memory locations.
   sv3 = (pfVec3 *)pfMalloc((int)num_verts * sizeof(pfVec3), arena);
   //Fill new array with all primitives' independent vertices' values.
   for (int g = 0; g<num_verts;  g++)
   {
     sv3[g][0] = sv2[svi2[g]][0];
     sv3[g][1] = sv2[svi2[g]][1];
     sv3[g][2] = sv2[svi2[g]][2];
   }
   //Reassign vertices to pfGeoSet and nullify the indices parameter.
   pfGSetAttr (gset1, PFGS_COORD3, PFGS_PER_VERTEX, sv3, NULL);

  //****Next, the rest of the attributes' indices arrays are eliminated similarly****
```

Figure 19: Example of Obtaining and Modifying pfGeoSet Data

b. Tristrips to Independent Triangles

The next conversion step provides more polygons to explode than are initially assigned to the pfGeoSet. This part converts tristrip primitives into the individual triangles of which they are comprised, if indeed the pfGeoSet is composed of tristrips (otherwise this part is skipped). The importance of this algorithm lies in the fact that almost all geometry is composed of tristrips since they are drawn very efficiently (as described in Chapter IV) due to the fact that vertices are shared among the triangles that compose the tristrip. As stated previously however, sharing of vertices does not allow for independent polygon motion.

Converting tristrips to triangles (Performer calls them tris) involves initially calculating the total number of vertices needed to assign each triangle its own set of vertices. This conversion must be implemented for all tristrip primitives contained in the pfGeoSet. The amount of triangles within a tristrip is equal to the number of vertices minus two and since a triangle has three vertices, the total number of vertices needed for independent triangles is equal to this amount of triangles multiplied by three. A new vertices array is created using this end result as its size to which all of the independent triangles' vertices positions are stored. However, this assignment is not trivial. The algorithm assigns vertices to the new array in sets of three. The first three vertices of the first tristrip primitive from the original array are assigned to the new array in that order as its first three values (the first triangle). The new array is then assigned its next three vertices in such a way that the first of these three (memory location three) is assigned the second to last vertex assigned in its previously assigned set of three vertices. The second of these three is assigned the last vertex assigned in its previously assigned set of three vertices. And finally, the third one is assigned the next vertex in the original tristrip array. Thus, this assignment simulates the tristrip drawing procedure as described in Chapter IV and therefore, continues in this manner until all triangles of the current tristrip are independent primitives. At this point, the algorithm must start the assignment process all over again with the pfGeoSet's next tristrip polygon. Starting over again means assigning to the new array

as its next three vertices, the first three vertices of the new tristrip primitive (which means the next three values in the original array) and NOT the previous last two from the three vertices most recently assigned to the new array. This overall algorithm continues until all tristrips belonging to the pfGeoSet have been converted to independent triangles and the pfGeoSet is "reset" with the new array of vertices positions. This algorithm is outlined in Figure 20.

```
if (prim_type == PFGS_TRISTRIPS)
{
   //******Calculate all tristrips' vertices and triangles.**********
   for (int z =0; z<num_prims; z++)
      {num_tris += lengths[z] - 2;      //triangles in a primitive = vertices - 2    }
   num_verts = num_tris * sides3;
   //**********Assign vertices for triangles to pfGeoSet.*********
   //*****Each Triangle is assigned the previous 2 vertices and the next vertex.*****
   //Allocate memory so that all triangles' vertices have separate memory locations.
   sv4 = (pfVec3 *)pfMalloc((int)(num_verts) * sizeof(pfVec3), arena);
   int rep = 0;              //Used to obtain the last two vertices of the previous triangle.
   int prevA = 0;            //Used to hold old array position where new tristrip starts.
   for (int j = 0; j<num_prims; j++)
   {
      for (int a = prevA; a<(((lengths[j]-2)*3) + prevA); a+=3)
      {
         sv4[a][0] = sv3[a-rep][0];
         sv4[a][1] = sv3[a-rep][1];
         sv4[a][2] = sv3[a-rep][2];
         sv4[a+1][0] = sv3[a-rep+1][0];
         sv4[a+1][1] = sv3[a-rep+1][1];
         sv4[a+1][2] = sv3[a-rep+1][2];
         sv4[a+2][0] = sv3[a-rep+2][0];
         sv4[a+2][1] = sv3[a-rep+2][1];
         sv4[a+2][2] = sv3[a-rep+2][2];
         rep +=2;
      }  //end inner "for"
      prevA = a;
      rep -= 2;                  //Starting with new tristrip, so don't need last two vertices.
   }//end outer "for"
   //Reassign vertices to pfGeoSet.
   pfGSetAttr (gset1, PFGS_COORD3, PFGS_PER_VERTEX, sv4, NULL);
```

Figure 20: Algorithm Used to Convert Tristrips to Independent Triangles

c. *Reassigning Tristrip Normals*

Not only do the tristrip pfGeoSet's vertices have to be reassigned, but the normals list must be changed to match the new independent triangles. Although normals' values can be assigned so that they are bound per vertex, per primitive, or overall, this function requires that they be assigned per primitive (one per triangle). The per primitive binding is necessary for the motion equations that follow the conversion algorithm in the explosion function and are explained later in this chapter. As illustrated in Figure 21, a triangle's normal is computed by treating two of its sides as vectors (X, Y, and Z values for each vertex are retrieved from the new array.), taking the cross product of these vectors, and then normalizing the result of the cross product. This computation is performed for all of the triangles after which an array the size of the total number of these triangles is created and assigned these normals which is then used to reset the pfGeoSet.

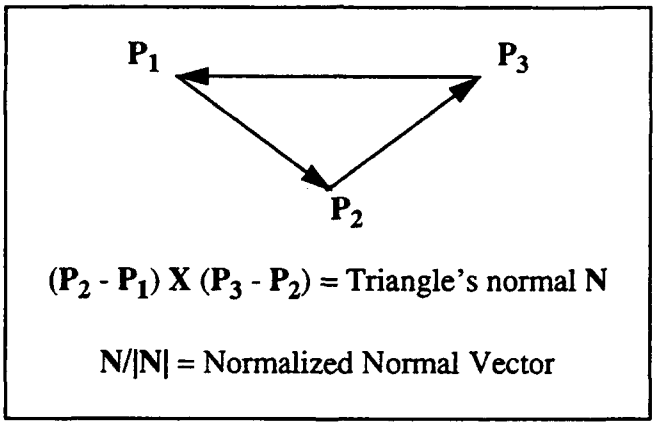

Figure 21: Computing the Normal of a Triangle

d. *Modification of Tristrip Textures and Colors*

Besides the normals, the remaining two construction attributes, textures and colors, must be modified to coincide with the new triangles. This part of the algorithm modifies these two attributes so that they are consistent with their original tristrip binding types.

Incorporating textures within a pfGeoSet means specifying coordinates that map the texture on to the object so as to cover it entirely or partially. Remapping a texture to a pfGeoSet which has been modified to include many more primitives than it once had, requires a complex computationally intensive algorithm in order to maintain the appearance of the overall object. In addition, when an explosion occurs it usually surprises a viewer and it lasts a very short amount of time meaning that texturing is not an attribute that is readily noticed or missed. Therefore, the conversion algorithm not only ignores texturing, but also deletes it. The reason for this deletion, is that texture mapping coordinates meant for one set of primitives are applied to a different group of primitives unpredictable and inconsistent results occur.

However, color conversion is accurately handled. If colors are bound to the pfGeoSet as a whole, then no conversion is needed. But, if they are assigned per primitive or per vertex then a conversion routine must be employed which is very similar to the tristrip to independent triangle conversion algorithm. The color per vertex reassignment routine is indeed exactly identical to the triangle conversion algorithm except that it modifies (expands) the color vice the vertices array. The color per primitive conversion is a little less complex, for it creates an array large enough to hold RGBA color components for each of the triangles and then assigns a color to the list for each triangle based on its original owning tristrip primitive. Therefore, triangles that are part of the same tristrip primitive are assigned the same color. Thus, the end result of this routine and the other color conversion algorithms is that the object's coloring remains consistent with the original pfGeoSet when tristrips are converted to independent triangles.

e. Completion of Conversion Algorithm

Finally after all pfGeoSet attributes have been converted, the only other aspects related to the pfGeoSet that need to be reset are the number and type of primitive that comprise the pfGeoSet. This assignment is accomplished by simply calling the Performer functions provided specifically for this purpose (pfGSetNumPrims,

pfGSetPrimType). Although these last two function calls complete the overall conversion of tristrips to triangles, there is one more part to the conversion algorithm that is employed if the pfGeoSet parameter is not made of tristrips.

As stated earlier pfGeoSets comprised of converted triangles must have normals that are bound per primitive to support the motion equations. This requirement also pertains to all other primitives which in the case of the explosion algorithm only include quads and original independent triangles (tris) - other Performer primitives are not capable of being filled with colors and therefore, are not handled. If normals are bound in any other way then this part of the conversion routine ensures that a pfGeoSet's normals array is permanently changed so that every primitive has one normal only. The method of modification used is the same as the one used for converted tristrips.

Finally, just as the normals' modification is permanent so are all of the rest of the pfGeoSet changes that have been presented. Therefore, conversions, if needed, are only performed during the first call to this function which is important, because these conversions require a significant amount of computation that if always implemented will slow down the explosion sequence. Thus, during the rest of the calls to the function the conversion algorithm is skipped and only the motion equations are applied to each of the polygons.

2. Explosion Motion Equations

The motion calculations used in this function are based on several physically-based concepts (assumptions) which are described in the following statements:

- When an object is impacted by a force, that force attenuates as the distance from the point of impact increases.
- When a force strikes an object, the point of impact is its center of mass.
- When a force impacts an object, that object rotates about the axis normal to the direction of the force.
- When an object is struck by a force it must overcome internal forces to break apart. (e.g., A brick building explodes into fragments if the force is strong enough to break the cement bond.)

a. Instantaneous Velocity Calculation

The motion algorithm's equations rely on the values generated in the previous call to the function except at the time of the first call. When the motion equations are executed at this time, the algorithm begins by calculating the initial impulse of the force applied to each of the polygons which is incorporated as the polygon's instantaneous velocity. This initial velocity is based on how close the polygon is to the point of impact as described in the concepts outlined above and it is thus, computed separately for each of the primitives using Equation 4 below where V_i is the velocity for the ith primitive.

$$\text{Impulse} = \text{Vi} = \text{deltatime} \times (\text{force} / (1 + \text{dist})) - \text{mass} \times k) \qquad \text{(Eq 6.1)}$$

Overall, the equation represents the instantaneous force multiplied by a time variable, deltatime. The variables in the equation that comprise the initial force inflicting the polygon at hand denote the following values: force is the initial force magnitude at the actual point of impact which is passed in as a parameter, dist is the distance from the center of mass of the polygon to this collision point, mass is the mass of the primitive which is a constant containing a value of ten kilograms, and k is also a constant set at 0.001. The reason that k is included is so that the equation takes into account the amount of force necessary to release the polygon from the bonding of the overall structure. This concept is incorporated by subtracting k multiplied by the variable mass from the rest of the equation. In addition, one is added to dist to prevent division by zero in the case that the point of impact is exactly at the polygon's center of mass. Finally, once the instantaneous velocity is computed for each of the polygons, the primitives' force direction vectors are multiplied by the corresponding initial velocity values. This scaling of the direction vectors creates the velocity vectors that are stored in the function's velocity array parameter which is continually manipulated by the ballistic motion equations during each call to this function (later in the algorithm), as previously mentioned. Figure 22 illustrates the preliminary computations needed to calculate the instantaneous velocity (polygons' centers of mass and force directional vectors) followed by the velocity's computation and its application to the

force direction vector. The example code presented in this figure is based on a pfGeoSet comprised of independent triangles. Additionally, Performer functions are used to perform the tasks that their names (after the pf) specify and return the result in their first arguments. Specifics regarding these functions are found in [SGIB94].

```
//*****Compute velocities for all polygons -> num_prims.*****
for (int i = 0; i<num_prims; i++)
{
   //*****Calculate the midpoint (center of mass) of each polygon.*****
   midptX = (sv4[sides3 * i][0] + sv4[sides3 * i+1][0] + sv4[sides3 * i+2][0])/sides3;
   midptY = (sv4[sides3 * i][1] + sv4[sides3 * i+1][1] + sv4[sides3 * i+2][1])/sides3;
   midptZ = (sv4[sides3 * i][2] + sv4[sides3 * i+1][2] + sv4[sides3 * i+2][2])/sides3;
   //*****Compute distance between polygon and point of impact.*****
   pfSetVec3(midpt, midptX, midptY, midptZ);
   dist = pfDistancePt3(where, midpt)      //where holds the point of impact
   //*****Compute force direction - polygon explosion path.*****
   force_dir[0] = midptX - where[0];
   force_dir[1] = midptY - where[1];
   force_dir[2] = midptZ - where[2];
   pfNormalizeVec3(force_dir);
   //*****Compute initial velocity one time per pfGeoSet polygon.*****
   if (!call_num)
   {
      //*****First assign Vel the force direction, then scale by initial velocity value.*****
      pfCopyVec3(Vel[i], force_dir);
      Vinit = deltatime * ((force/(1+dist))- mass*k;
      pfScaleVec3(Vel[i], Vinit, Vel[i]);
   } //end if !call_num
//*****The for loop continues to include the rest of the motion equations.*****
```

Figure 22: Computation of Instantaneous Velocity for Triangle Primitives

a. Polygons' Rotation

Another aspect of the polygons' movement that is calculated by this function is the direction of each polygon's rotation. This direction is computed by taking the cross product of a primitive's force vector (not scaled by the velocity magnitude described above) with its normal, and then normalizing the result. Thus, this is the point in the motion algorithm where having a normal bound per primitive is important. Otherwise, time consuming normal calculations would have to be computed during every call of the explosion sequence. The rotational amount, however, is not calculated, instead all

primitives are rotated about their computed axes ten degrees after every call to the explosion function.

b. Ballistic Motion Equations

Once the rotation directions have been calculated the only computation left to perform is the polygon's actual translation along its force vector. Ballistic motion equations, specifically Cromer-Euler equations which were chosen due to their speed and ability to correct their own inaccuracies periodically, are used to model this translation. These computations employ both gravity and an air friction coefficient whose values are constant in this function - 9.8 meters per second squared and 2.0 respectively. As previously mentioned, the velocity initially used in these equations is the resultant velocity from the preceding call to this function (unless the function has been called for the first time for the pfGeoSet at hand). These velocity values are used to compute the new values for the force components (X, Y, and Z), which are used to compute the acceleration components of the primitives, and finally the acceleration components are multiplied by the deltatime parameter to produce the new velocity components. These ballistic motion computations are displayed in Figure 23 where Vel holds the velocity vectors for all primitives of the pfGeoSet.

```
//*****Compute polygon's position with ballistic motion equations.*****
//(Continuation of for loop that cycles through all pfGeoSet primitives.)
forceX = (-1.0) * air_fric * Vel[i][0];
forceY = (-1.0) * air_fric * Vel[i][1];
forceZ = (-1.0) * mass * gravity - air_fric * Vel[i][2];

accelX = forceX/mass;
accelY = forceY/mass;
accelZ = forceZ/mass;

Vel[i][0] = accelX * deltatime;
Vel[i][1] = accelY * deltatime;
Vel[i][2] = accelZ * deltatime;
```

Figure 23: Ballistic Motion Equations

c. *Completion of Motion Algorithm*

The last function this algorithm performs is the actual changing of each primitive's location values in memory to reflect their newly calculated rotations and positions since the last call to the explosion function. These new positions are the result of multiplying each primitive's velocity by the deltatime argument. Thus, after this function returns control to the calling program, that application's draw process displays the primitives with their new orientation at their new locations.

C. PERFORMANCE WITHIN A TEST APPLICATION

In order to test the explosion function we developed a simple application that loads into a Performer world one or more objects that a user may shoot and then witness their explosions. The user moves through this environment by pushing the arrow keys on the keyboard. When the viewer's cross-hairs displayed on the screen are positioned in front of an object, the operator is able to shoot it by pushing the "S" key. An intersection routine is called when this key is pressed to determine if an entity in the scene is in front of and close enough to the viewer to have been hit by this weapon (within 5000 meters). If an object was hit, then this routine returns a pointer to the actual pfGeoSet that was struck (even if the object was originally built outside of the Performer environment), the exact point of impact position, and a flag indicating that a collision did occur. Upon the receipt of these values, the application proceeds to call the explosion routine every time through its simulation loop with a number of calls counter, the force (which is a constant equaling 1000 newtons), the pfGeoSet pointer, an array allocated to hold the velocity of the exploding polygons belonging to the pfGeoSet, the time it took to get through the simulation loop, and the point of impact as the parameters to the function. Once the number of calls counter (which is incremented in the explosion function every time that it is called) reaches 150 the application stops calling the explosion function for that pfGeoSet.

After designing this application which verified that explosions produced by the explosion function occur realistically, we ran some tests to determine the function's

potential for incorporation in NPSNET. Our first test involved shooting a building made of seven quad primitives where no conversion to explodable format except for rebinding the normals to per primitive was necessary. During the explosion, the frame rate remained constant at its preset rate of 30 frames per second. In our next test the explosion function had to convert a helicopter model made of 89 tristrip primitives (all contained either one or two triangles) to explodable format before the polygons were discharged. The total triangles resulting from this conversion was 160 and again the frame rate remained constant at 30 frames per second except for an initial flash indication of a frame rate of 20 frames per second. This latter frame rate was due to the tristrip conversion process. Our last test involved part of a building structure which was comprised of 127 tristrip primitives (again, all contained either one or two triangles) that had to be converted. The conversion resulted in 230 triangles and with this increased amount of polygons, the explosion only operated at a rate of 15 frames per second. An example of a "before and after shot" is illustrated in Figures 24 and 25 which display the helicopter before the weapon's impact and then during the explosion respectively.

Figure 24: Helicopter Before Impact

Figure 25: Helicopter After Impact

D. EXPLOSIONS SUMMARY

The performance tests above indicate that explosions occur in real-time on their own; however the amount of calculations involved in the function's algorithms are too many to be handled by a real-time synthetic environment like NPSNET, especially if many explosions are occurring simultaneously. There are however, minor modifications that could be made to facilitate their use within NPSNET. One such modification entails performing all of the function's computations except the actual ballistic motion equations prior to run-time and placing the values in a lookup table(s). This implementation would require that the simulation have prior knowledge of all possible objects that could exist in its world and would take away from the function's ability to handle any entity that it is sent. Another way that these explosions might be implemented within NPSNET, is by having a an "explodable" model available to switch in to the scene for every object that could rupture. Thus, when an object is struck, its "alive" model would be switched out and replaced with an explodable model that would be erupted by the function. In addition, no conversions would be necessary and the object could be comprised only of an amount of

polygons that would be suitable for exploding within a real-time simulator. Overall, these ideas are just a couple of options that could be implemented without detracting from the physically-based aspects of the explosions. However, integration of these effects within NPSNET may require incorporating a variation of these ideas or more drastic revisions resulting in less realistic explosive effects.

VII. CONCLUSION AND TOPICS FOR FURTHER RESEARCH

A. CONCLUSION

Providing a truly believable virtual battlefield simulator means that designers must take into consideration a wide variety of occurrences that are both obviously present in a battle zone and those that are less apparent. Now that the ground work for such combat simulators has been laid, many researchers are realizing the significant role that these less obvious effects play in providing a truly convincing synthetic environment. It has also become apparent just how many of these effects exist and the amount of detail required to realistically model them. In addition, networking further complicates this issue. In light of these circumstances, researchers have found it difficult to incorporate such effects while maintaining their simulator's real-time performance. This problem is due in part to the limited amount of computation power available. Our work addresses some of these effects and their potential for incorporation within such simulators like NPSNET. We too ran into these difficulties, particularly in terms of providing the ability to fly through realistic clouds and in modeling cloud movement based on gridded wind vectors. However, we have provided a starting point for incorporating our effects in real-time simulators by designing a variety of test harnesses (as have been previously described) for studying and experimenting with modifications to these models. Thus, these mini-simulators provide feedback regarding models' potential for incorporation within dynamic real-time virtual worlds.

B. FUTURE RESEARCH

Future research in the area of modeling dynamic effects is vast. In terms of meteorological occurrences, frontal systems, precipitation, and flooding need to be simulated. Dynamic terrain issues requiring future research include implementing areal features (such as oil spills and rivers) and modeling actual modifications to the terrain database as opposed to placing earthworks such as craters or berms on top of the terrain. These terrain modifications have been simulated, but they do not exist in real-time

simulators like NPSNET. Lastly, in the area of explosive effects, all forces including air friction and the object's internal bonding forces need to be accurately determined and properly applied to the exploding objects.

APPENDIX. USER'S GUIDES

This appendix contains all of the information required to run the set of effects simulators described in this thesis. These simulators include the Cloud Modeler, wind vector simulator, DIS Destructible Entity PDU Implementation Program, and explosions test harness.

A. CLOUD MODELER

1. How to Start the Program

The Cloud Modeler is started by entering the following on the command line where filename is a file that exists in the current directory for writing cloud data to:

cloudmodeler -w filename

2. Input Devices for Designing and Viewing a Cloud

a. Mouse

Move Mouse Right - User turns right.

Move Mouse Left - User turns left.

Push Left Button - User accelerates forward.

Push Right Button- User accelerates backward.

b. Key Bindings

The key bindings below indicate the order in which the keys need to be pressed in order to perform the modification represented by the sequence. Keys are to be pressed sequentially not all at once.

ESCAPE KEY - Exits the program.

Q KEY - Turns on Performer's pfDrawChanStats for evaluating program's performance. See [SGIA94] for a complete description of this function.

W KEY - Writes current cloud's parameters to the designated file.

R KEY+A KEY+MINUS KEY - Decrements cloud's red ambient component by 0.1.

R KEY+A KEY+PLUS KEY -Increments cloud's red ambient component by 0.1.

G KEY+A KEY+MINUS KEY - Decrements cloud's green ambient component by 0.1.

G KEY+ A KEY +PLUS KEY - Increments cloud's green ambient component by 0.1.

B KEY+A KEY+MINUS KEY - Decrements cloud's blue ambient component by 0.1.

B KEY+A KEY+PLUS KEY - Increments cloud's blue ambient component by 0.1.

R KEY+D KEY+MINUS KEY - Decrements cloud's red diffuse component by 0.1.

R KEY+D KEY+PLUS KEY - Increments cloud's red diffuse component by 0.1.

G KEY+D KEY+MINUS KEY - Decrements cloud's green diffuse component by 0.1.

G KEY+D KEY+PLUS KEY - Increments cloud's green diffuse component by 0.1.

B KEY+D KEY+MINUS KEY - Decrements cloud's blue diffuse component by 0.1.

B KEY+D KEY+PLUS KEY - Increments cloud's blue diffuse component by 0.1.

T KEY+MINUS - Decrements cloud's opacity component by 0.1.

T KEY+PLUS - Increments cloud's opacity component by 0.1.

O KEY - Displays cloud with just the texture applied - no ambient, diffuse, or opacity components are incorporated.

S KEY+F1 KEY- Scales cloud's size by a factor of 10.

S KEY+F2 KEY- Scales cloud's size by a factor of 20.

S KEY+F3 KEY- Scales cloud's size by a factor of 30.

S KEY+F4 KEY- Scales cloud's size by a factor of 40.

S KEY+F5 KEY- Scales cloud's size by a factor of 50.

S KEY+F6 KEY- Scales cloud's size by a factor of 60.

S KEY+F7 KEY- Scales cloud's size by a factor of 75.

S KEY+F8 KEY- Scales cloud's size by a factor of 100.

S KEY+F9 KEY- Scales cloud's size by a factor of 150.

S KEY+F10 KEY- Scales cloud's size by a factor of 200.

C KEY+F1 KEY - Displays 1 puff.

C KEY+F2 KEY - Displays 2 puffs.

C KEY+F3 KEY - Displays 3 puffs.

C KEY+F4 KEY - Displays 30 puffs.

C KEY+F5 KEY - Displays 40 puffs.

C KEY+F6 KEY - Displays 50 puffs.

C KEY+F7 KEY - Displays 75 puffs.

C KEY+F8 KEY - Displays 100 puffs.

C KEY+F9 KEY - Displays 150 puffs.

C KEY+F10 KEY - Displays 200 puffs.

U KEY+1,2,3,4,5,6,7,8,9, or 0 KEY+F1 KEY - Displays cloud with uniform distribution of puffs in 1,2,3,4,5,6,7,8,9, or 10 row(s) and with a depth of one.

U KEY+1,2,3,4,5,6,7,8,9, or 0 KEY+F3 KEY - Displays cloud with uniform distribution of puffs in 1,2,3,4,5,6,7,8,9, or 10 row(s) and with a depth of three.

U KEY+1,2,3,4,5,6,7,8,9, or 0 KEY+F5 KEY - Displays cloud with uniform distribution of puffs in 1,2,3,4,5,6,7,8,9, or 10 row(s) and with a depth of five.

U KEY+1,2,3,4,5,6,7,8,9, or 0 KEY+F10 KEY - Displays cloud with uniform distribution of puffs in 1,2,3,4,5,6,7,8,9, or 10 row(s) and with a depth of ten.

N KEY+F1 KEY - Displays cloud with a normal distribution of puffs with a depth of one.

N KEY+F3 KEY - Displays cloud with a normal distribution of puffs with a depth of three.

N KEY+F5 KEY - Displays cloud with a normal distribution of puffs with a depth of five.

N KEY+F10 KEY - Displays cloud with a normal distribution of puffs with a depth of ten.

Z KEY+F1 KEY - Displays cloud with a symmetrical normal distribution of puffs with a depth of one.

Z KEY+F3 KEY - Displays cloud with a symmetrical normal distribution of puffs with a depth of three.

Z KEY+F5 KEY - Displays cloud with a symmetrical normal distribution of puffs with a depth of five.

Z KEY+F10 KEY - Displays cloud with a symmetrical normal distribution of puffs with a depth of ten.

L KEY+R KEY+M KEY+MINUS KEY - Decrements light's red ambient component by 0.1. (M for ambient)

L KEY+R KEY+M KEY+PLUS KEY -Increments light's red ambient component by 0.1.

L KEY+G KEY+M KEY+MINUS KEY - Decrements light's green ambient component by 0.1.

L KEY+G KEY+M KEY+PLUS KEY - Increments light's green ambient component by 0.1.

L KEY+B KEY+M KEY+MINUS KEY - Decrements light's blue ambient component by 0.1.

L KEY+B KEY+M KEY+PLUS KEY - Increments light's blue ambient component by 0.1.

L KEY+R KEY+H KEY+MINUS KEY - Decrements light's red diffuse component by 0.1. (H for hue)

L KEY+R KEY+H KEY+PLUS KEY - Increments light's red diffuse component by 0.1.

L KEY+G KEY+H KEY+MINUS KEY - Decrements light's green diffuse component by 0.1.

L KEY+G KEY+H KEY+PLUS KEY - Increments light's green diffuse component by 0.1.

L KEY+B KEY+H KEY+MINUS KEY - Decrements light's blue diffuse component by 0.1.

L KEY+B KEY+H KEY+PLUS KEY - Increments light's blue diffuse component by 0.1.

L KEY+P KEY+PLUS KEY - Light comes from a quadrant in X-Y plane one higher than the previously current one (quadrants I, II, III, IV). If the current one is quadrant IV then this key sequence will cause the light to come from quadrant I.

L KEY+P KEY+MINUS KEY - Light comes from a quadrant in X-Y plane one less than the previously current one (quadrants I, II, III, IV). If the current one is quadrant I then this key sequence will cause the light to come from quadrant IV.

R KEY+Y KEY+MINUS KEY - Decrements light model's red ambient component by 0.1.

R KEY+Y KEY+PLUS KEY -Increments light model's red ambient component by 0.1.

G KEY+Y KEY+MINUS KEY - Decrements light model's green ambient component by 0.1.

G KEY+Y KEY+PLUS KEY - Increments light model's green ambient component by 0.1.

B KEY+Y KEY+MINUS KEY - Decrements light model's blue ambient component by 0.1.

B KEY+ Y KEY+ PLUS KEY - Increments light model's blue ambient component by 0.1.

B. WIND VECTOR SIMULATOR

1. How to Start the Program

The wind vector simulator is started by entering the following on the command line where filename is a file that exists in the current directory and was written to by the Cloud Modeler:

winds -r filename

2. Input Devices for Viewing the Clouds Within the Wind Vector World

a. Mouse

Move Mouse Right - User turns right.

Move Mouse Left - User turns left.

Push Left Button - User accelerates forward.

Push Right Button- User accelerates backward.

b. Key Bindings

The key binding below present various views of the world. Figure 11 in Chapter 4 presents the actual winds that are being displayed.

ESCAPE KEY - Terminates program.

F1 KEY - Turns on Performer's pfDrawChanStats for evaluating program's performance. See [SGIA94] for a complete description of this function.

F2 KEY - Presents user with a view of the boundary between cells 2 and 3 where wind vectors directly oppose each other. The boundary between cells 3 and 4 is also displayed where there is a change in velocity.

F3 KEY - Presents user with a view of the boundary between cells 6and 7 where wind vectors directly oppose each other.

F4 KEY - Presents user with a view of the boundary between cells 5 and 6 where 30^{o} difference in headings exists.

F5 KEY - Present two boundaries where changes in velocities occur - between cells 0 and 1 and between cell 1 and 2.

B KEY - Presents an aerial view of the entire world. User must move mouse downward to initially view. Also, removes user's height flight restriction (0.0 m to 10000.0 m in X, 0.0 m to 10000.0 m in Y, and 3.0m to 8000.0 in Z). If this key is pressed again, viewer is back in the world.

F KEY - Removes displayed cell boundaries. If this key is pressed again, cell boundaries are displayed.

C. DIS DESTRUCTIBLE ENTITY PDU IMPLEMENTATION PROGRAM

This 2D grid modeler implementation of DIS Destructible Entity PDUs is run on one machine as the terrain manger and on one or more other machines as the simulation participant(s). The terrain manager needs to be started first so that it does not miss any world occurrences. Host players may be started at any time after the terrain manager initialized. Both the participants' and terrain manager's programs are started by simply typing run on the command line. In addition the following key bindings exist:

ESCAPE KEY - Terminates the terrain manager's or player's program.

END KEY - Pressing this key on a player's host begins the periodic shooting of missiles by that participant.

DELETE KEY - Resets the player's launcher.

A KEY - Allows user of the terrain manager to send a test Create/Modify PDU without the receipt of a Detonation PDU.

D KEY - Allows the user of the terrain manager to send a test Deletion PDU.

D. EXPLOSIONS TEST HARNESS

1. How to Start the Program

This test harness is simply run by typing on the command line:

blowup [-p x y z] [-f flightfile] <-p x y z> <-f flightfile> ...

[ ] = required input

< > = optional input - multiple sets of above two <>'s may be specified, but they must occur in sets.

-f = indicates a Multigen file follows that is to be loaded by the program

flightfile = specifies the Multigen file to be loaded

-p = indicates the position of the Multigen object follows

x y z = the position in the world (eyepoint always starts at 0,0,0)

2. Key Bindings

ESCAPE KEY - Terminate the program.

F1 KEY - Turns on Performer's pfDrawChanStats for evaluating program's performance. See [SGIA94] for a complete description of this function.

PAGE UP KEY - Increases viewer's eye elevation position.

PAGE DOWN KEY - Decreases viewer's eye elevation position.

UP ARROW KEY - User moves forward one unit in direction of view.

DOWN ARROW KEY - User moves backward one unit in direction of view.

RIGHT ARROW KEY - User moves right one unit.

LEFT ARROW KEY - User moves left one unit.

KEYPAD 8 KEY (up) - Increases user's direction of view pitch up one unit.

KEYPAD 2 KEY (up) - Decreases user's direction of view pitch down one unit.

KEYPAD 6 KEY (up) - Increases user's direction of view heading right one unit.

KEYPAD 8 KEY (up) - Decreases user's direction of view heading left one unit.

S KEY - Shoots missile.

LIST OF REFERENCES

[BOEH94] Boehm, A., "Visual Translucent Algorithm (Vista)," *Simulation*, February 1994.

[BURG91] Burg, J., Moshell, J., et al., Behavioral Representation in Virtual Reality. *Proceedings of Behavioral Representation Symposium.* Institute for Simulation and Training. Orlando, Florida, 1991.

[CORB93] Corbin, D., *NPSNET: Environmental Effects for a Real-Time Virtual World Battlefield Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

[FAA65] Federal Aviation Agency and Department of Commerce, "*Aviation Weather*," by W. Nash, 1965.

[IST93] Institute for Simulation and Training, IST-CR-93-15, *Standard for Information Technology - Protocols for Distributed Interactive Simulation Applications [Proposed IEEE Standard Draft]*, University of Central Florida, Orlando, Florida, May 1993.

[ISTA93] Institute for Simulation and Training, IST-TR-93-10, *Distributed Interactive Simulation Operational Concept [Draft 2.2]*, University of Central Florida, Orlando, Florida, March 1993.

[IST94] Institute for Simulation and Training, *Frequently Asked Questions About Dynamic Terrain*, University of Central Florida, Orlando, Florida, 1994.

[LI93] Li, X., Moshell, J., "Modeling Soil: Realtime Dynamic Models for Soil Slippage and Manipulation," *Computer Graphics Proceedings*, Annual Conference Series 1993, ACM SIGGRAPH 1-6 August 1993.

[LIES66] Lieske, R., and Reiter, M., "Equations of Motion for a Modified Point Mass Trajectory," *Ballistics Research Laboratories* Report Number 1314, March 1966.

[LISL94] Lisle, C., Altman, M., Kilby, M., Sartor, M., "Architectures for Dynamic Terrain and Dynamic Environments in Distributed Interactive Simulation," *10th DIS Workshop on Standards for the Interoperability of Defense Simulations, Volume II Position Papers*, March 1994.

[LADS94] Loral Advanced Distributed Simulation, "*Final Report Dynamic Environment Modeling in Distributed Interactive Simulations*," Loral Advanced Distributed Simulation, May 1994.

[MONA91] Monahan, J., *NPSNET: Physically-Based Modeling Enhancements to an Object File Format*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1991.

[NASH92] Nash, D., *NPSNET: Modeling the In-Flight and Terminal Properties of Ballistic Munitions*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1992.

[NAVA79] Navarra, J., "*Atmosphere, Weather and Climate: An Introduction to Meteorology*," W.B. Saunders Company, Philadelphia, Pennsylvania, 1979.

[SCHA81] Schaefer, V. and Day, J., "*A Field Guide to the Atmosphere*," Houghton Mifflin Company, Boston, Massachusetts, 1981.

[SGIA94] Silicon Graphics, Inc. Document Number 007-1680-020, *IRIS Performer Programming Guide*, J. Hartman and P. Creek, 1994.

[SGIB94] Silicon Graphics, Inc. Document Number 007-1681-020, *IRIS Performer Reference Pages*, S. Fischler, J Helman, M. Jones, J. Rohlf, A. Schaffer, and C. Tanner, 1994.

[THOR87] Thorp, J., "*The New Technology of Large Scale Simulator Networking: Implications for Mastering the Art of Warfighting*," Proceedings of the 9th Interservice/Industry Training System Conference, November - December 1987.

[WALT92] Walters, A., *NPSNET: Dynamic Terrain and Cultured Feature Depiction*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1992.

[WEIS91] Weiss, N. and Hassett, M., "*Introductory Statistics*," 3rd Edition, Addison-Wesley Publishing Company Inc., Menlo Park, California, 1991.

[ZYDA90] Zyda, M., *Naval Postgraduate School Graphics and Video Laboratory Notes, Book 6*, 1990.

[ZYDA93] Zyda, M., Pratt, D., Falby, J., Barham, P., and Kelleher, K., "NPSNET and the Naval Postgraduate School Graphics and Video Laboratory," *Presence*, Vol. 2, No. 3, 1993.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 052
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Dr. Ted Lewis, Chairman and Professor 1
   Computer Science Department Code CS/TL
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. David R. Pratt, Assistant Professor 2
   Computer Science Department Code CS/PR
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Dr. Michael J. Zyda, Professor 2
   Computer Science Department Code CS/ZK
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Mr. Paul Barham, Computer Specialist 1
   Computer Science Department Code CS
   Naval Postgraduate School
   Monterey, CA 93943-5000

7. U.S. Army Topographic Engineering Center 1
   Attn: Jeff Turner CETEC-TD-SM
   7701 Telegraph Road
   Alexandria, VA 22315-3864

8. Lt. Anne E. Watt USN 2
   Computer Science Dept. 9F
   U.S. Naval Academy
   572 Holloway Rd.
   Annapolis, MD 21402