

1-1-1995

Dynamic Terrain: Prototype Software Documentation

Glenn A. Martin

Find similar works at: <https://stars.library.ucf.edu/istlibrary>
University of Central Florida Libraries <http://library.ucf.edu>

This Research Report is brought to you for free and open access by the Digital Collections at STARS. It has been accepted for inclusion in Institute for Simulation and Training by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

Recommended Citation

Martin, Glenn A., "Dynamic Terrain: Prototype Software Documentation" (1995). *Institute for Simulation and Training*. 222.
<https://stars.library.ucf.edu/istlibrary/222>



INSTITUTE FOR SIMULATION & TRAINING

DYNAMIC TERRAIN PROTOTYPE SOFTWARE DOCUMENTATION

Contract Number N61339-92-K-0001, N61339-94-C-004
US ARMY STRICOM

July 14, 1995
IST-TR-95-16



B327

DYNAMIC TERRAIN -

PROTOTYPE SOFTWARE

DOCUMENTATION

Contract Numbers N61339-92-K-0001, N61339-94-C-0004
US ARMY STRICOM

July 14, 1995
IST-TR-95-16

Authors:

Glenn Martin • Lance Marrou • Sean Waldron • Xiong Zhiguo • Jim Chen
Guru Prasad • Mark Kilby • Michelle Sartor • Curtis Lisle • Marty Altman

Technical Writers:

Melissa Hinds • Kim Nelson • Helen Clarke

Reviewed by:

Art Cortes

Mark Kilby

Marty Altman

Michelle Sartor

Art C. Cortes

Mark Kilby

Marty Altman

Michelle M. Sartor

Visual Systems Laboratory
Institute for Simulation and Training • 3280 Progress Drive • Orlando, FL 32826
University of Central Florida • Division of Sponsored Research

Table of Contents

Dynamic Terrain Shared Environment	1
Dynamic Terrain Database	2
Introduction to the DynamicTerrain Database	2
Goals	2
Assumptions about the Active Database Element	2
Analysis	3
The Abstract Form of the Query	4
Implementation	4
User's Guide	4
The Area Form of the Query	7
Programmer's Guide	11
 Abstract Service	 29
Introduction to the Abstract Service	29
Analysis	29
Service	29
Service Interface	30
Client Interface	30
Networking	30
Coordinate Conversion	30
Implementation	30
Appendix	30
One Central Server	30
A Collection of Group Servers	31
One Central Server with Group Relay Servers	31
Fully Distributed	32
 Entity Service	 33
Introduction to Entity Service	33
Background	33
Problem Statement (external)	34
Solution	34
Analysis	35
Implementation	35
Problem Statement	35
User's Guide	35
Software Design	47
Programmer's Guide	47
 Terrain Service	 51
Introduction to Terrain Service	51
Background	51
Problem Statement	51

Solution	52
Analysis	52
Implementation	53
Problem Statement.....	53
User's Guide	53
Software Design.....	62
Programmer's Guide.....	62
Fluid Service	66
Introduction to Fluid Service	66
Background	66
Problem Statement.....	66
Solution	67
Analysis	67
Implementation	68
Problem Statement.....	68
User's Guide	68
Software Design.....	76
Programmer's Guide.....	76
Dynamic Terrain Resources - Shared Environment Clients.....	80
Soil DTR	81
Introduction to Soil DTR	81
Background	81
Problem Statement.....	81
Solution	81
Constraints/Assumptions	81
Analysis	82
Preliminaries	82
Static Equilibrium and Restoring Force.....	83
Extension to 3-D	88
Implementation	89
Problem Statement.....	89
User's Guide	90
Software Design.....	91
Programmer's Guide.....	91
Fluid DTR	96
Introduction to Fluid DTR	96
Background	96
Problem statement.....	97
Solution	97
Constraints/assumptions about problem	98
Analysis	98
Real-time Fluid Model.....	98
Fluid Model in a DT Simulation.....	104

Implementation	107
Problem Statement.....	107
User's Guide	109
Software Design.....	111
Programmer's Guide.....	112
Crater DTR	113
Introduction to Crater DTR	113
Background.....	113
Problem Statement.....	113
Solution	113
Constraints/Assumptions	114
Analysis	114
Soil Displacement.....	114
Thermal Attribute.....	115
Implementation	115
Problem Statement.....	115
User's Guide	116
Software Design.....	116
Programmer's Guide.....	117
Track DTR	120
Introduction To the Track DTR	120
Background.....	120
Problem Statement.....	120
Solution	120
Constraints/Assumptions	120
Analysis	122
Instantaneous Track Points	122
Soil Displacement.....	123
Terrain Patch Query and Update.....	124
Implementation	125
Problem Statement.....	125
User's Guide	126
Software Design.....	130
Programmer's Guide.....	130
Thermal DTR	135
Introduction to Thermal DTR	135
Background.....	135
Problem Statement.....	135
Solution	135
Constraints/Assumptions	135
Analysis	136
Implementation	136
Problem Statement.....	136

User's Guide	136
Software Design.....	138
Programmer's Guide.....	138
Minefield DTR	139
Introduction to Minefield DTR	139
Background.....	139
Problem Statement.....	139
Solution.....	139
Constraints/Assumptions	139
Analysis	140
Minefield Model	140
Implementation	141
Problem Statement.....	141
User's Guide	141
Programmer's Guide.....	142
More Shared Environment Clients	143
Visualizer	144
Introduction To Visualizer	144
Background.....	144
Problem Statement.....	144
Solution.....	144
Constraints / Assumptions	145
Implementation	145
Problem Statement.....	145
User's Guide	145
Software Design.....	149
Programmer's Guide.....	149
IG Host	151
Introduction to the IG Host	151
Analysis	152
TVS Decisions	155
Performer was used for Flight Files.....	155
The TVS separated into managers for each major function	155
OOPS	156
Soil Sample Objects.....	156
Vehicle State and Vehicle Controls	156
Vehicle State and Controls Hierachical Structure	156
FORMS library used for the GUI	156
Command pipes were implemented.....	157
Reuse of Bwana Vision.....	157
The Entity Service Used	157
MVC Proposal	157
Simulation Overview	158

Diagram Description.....	159
Network Services	161
Entity Service.....	161
Terrain Service.....	161
Simulation Host	161
Graphical User Interface	162
FORMS.....	162
IRIS Performer.....	163
Image Generator Host	163
Image Generator	163
ESIG 2000.....	164
BwanaVision (BV).....	164
IG Host to IG to GUI Connectivity	164
Overview of Alternate Designs for the Simulation System.....	166
IG Host to IG Connectivity	171
Introduction.....	171
Entity Service (ES)	172
Terrain Service (TS)	172
Description of Alternative Designs of the Host-IG Interface	172
IG Design	179
Introduction.....	179
Entities	180
Dynamic Terrain	183
Segregation of Responsibilities.....	187
Recommendations.....	193
Data Flow Diagrams	194
IG Protocol	197
Introduction.....	197
What is IG Protocol?.....	199
Classes with Suggested Protocol Methods	200
The IG Host as a Stealth	203
Controlling the IG Host	204
Viewpoint Offsets	204
GUI Design	205
Introduction.....	205
GUI as a Separate Process	206
Proposed Design	207
Communication Description	209
GUI Protocol	213
GUI Service	213
Continuous State Data.....	214
Event Driven Data.....	214
ESIG Host	216
Introduction to the ESIG Host	216
Analysis	216

Entity State Message.....	216
Detonation Message.....	217
DT Attribute Message.....	217
Simulation Host	218
Introduction to Simulation Host	218
Background.....	218
Problem Statement.....	218
Solution.....	218
Constraints/Assumptions	220
Analysis	220
Implementation	221
Problem Statement.....	221
User's Guide	221
Software Design.....	225
Programmer's Guide.....	228
Dynamic Terrain Utilities	229
Dynamic Terrain Database Formatter	230
Introduction to the Dynamic Terrain Database Formatter	230
Problem Statement.....	230
Solution.....	230
Analysis	230
Implementation	231
Problem Statement.....	231
User's Guide	232
Examples.....	237
Surface Resampler	239
Introduction to Surface Resampler	239
Analysis	239
Implementation	239
User's Guide	239
Fractal Resampler	240
Introduction to Fractal Resampler	240
Analysis	240
Implementation	240
User's Guide	240
Terrain Service Configuration Utility	241
Introduction to the Terrain Service Configuration Utility	241
Problem Statement.....	241
Solution.....	241
Implementation	241
Problem Statement.....	241

User's Guide	242
Examples.....	244
Fluid Service Configuration Utility	248
Introduction to the Fluid Service Configuration Utility	248
Problem Statement.....	248
Solution	248
Implementation	248
Problem Statement.....	248
User's Guide	249
Example	251
Sources	254
Introduction to Sources	254
Problem Statement.....	254
Solution	254
Analysis	254
Implementation	255
Problem Statement.....	255
User's Guide	256
Programmer's Guide.....	262
Sinks	266
Introduction to Sinks	266
Problem Statement.....	266
Solution	266
Analysis	266
Implementation	266
Problem Statement.....	266
User's Guide	267
Example Output	269
Appendices	272
Mobility and Vehicle Dynamics	273
Introduction to Mobility and Vehicle Dynamics	273
Motivation for Mobility Studies	273
Definitions.....	273
Tracked Vehicles on Dynamic Terrain	274
Functional Requirements	274
Mathematical Modeling Requirements.....	274
Simulation Flow	275
Blade System	275
Vehicle System	277
Terra-Mechanics	278
Design & Analysis Methodology	278
Vehicle System	278

Vehicle Dynamics	278
Blade Mechanics	284
Soil Mechanics	286
Conclusions	290
Vehicle Performance Curve	292
Soil Attributes	293
Digging Tool Model	295
Introduction to the Digging Tool Model	295
Background	295
Problem Statement	295
Solution	295
Constraints/Assumptions	295
Analysis	296
Digging Tool Geometry	296
Soil Manipulation Procedure	298
Implementation	302
Problem Statement	302
User's Guide	303
Software Design	307
Programmer's Guide	307
Coordinate Conversion	312
Introduction to Coordinate Conversion	312
Types of Coordinate Conversions	312
Global Vector Conversions	312
Body Vector Conversions	313
Orientation Conversions	313
Layout of the Client Interface	313
Types of Coordinate Systems	314
Types of Database Coordinate Systems	314
Types of Entity/Event Coordinate Systems	316
Types of Orientation Formats	317
Coordinate Systems used in the Client Interface	317
Conversions between Coordinate Systems	318
Position and Global Vector Conversions	318
Body Vector Conversions	320
Conversion of Orientation	322
Dead Reckoning	325
Introduction	325
Dead Reckoning with DIS Standards	325
Kinematic Data Fields in DIS	327
Position	327
Orientation	327
Angular Velocity	327

Translational Velocity and Acceleration	328
Implementation of Dead Reckoning Models	328
Algorithm Zero, "Other" Model	328
Algorithm One, Static Model.....	328
Algorithm Two, Constant Global Velocity with Fixed Axes	328
Algorithm Three, Constant Global Velocity with Rotating Axes.....	329
Algorithm Four, Constant Global Acceleration with Rotating Axes.....	329
Algorithm Five, Constant Global Acceleration with Fixed Axes.....	330
Algorithms Six, Seven, Eight, and Nine	330
Algorithm Six, Constant Body Velocity with Fixed Axes	331
Algorithm Seven, Constant Body Velocity with Rotating Axes	331
Algorithm Eight, Constant Body Acceleration with Rotating Axes.....	332
Algorithm Nine, Constant Body Acceleration with Fixed Axes	334
Special Rotation Algorithms for Dead Reckoning	335
Generation of the Initial Rotation Matrix	335
Dead Reckoning of Orientation	336
Determining Euler Angles from a Rotation Matrix	337
Smoothing from Current to New States	339
Snapping State Values	339
Determining a Goal State.....	339
Linear Interpolation between States.....	340
Fitting a Hermite Spline between States.....	341
IG Host User's Manual	343
Introduction to the IG Host User's Manual	343
System Configuration Requirements	344
Project Management	344
Start-up/Configuration	344
Command Line Parameters.....	345
How to Use Dynamic Terrain	346
IG Host Configuration File (IGhost.cfg)	347
DracVision IG Configuration File (IG.cfg)	348
Run-Time Commands	349
IG Host Initialization File (IGhost.init)	349
Other IG Host Commands	352
Other DracVision Commands.....	353
DracVision Keyboard Commands	363
References	364

List of Figures

A Conceptual Model of the Data within the DynamicTerrainDatabase	5
The Client's View of the Area Form of the Query	7
The DynamicTerrainDatabase (less simple instance variables)	12
Translation of the Area Query into Point Queries	13
Surface Representations Class Hierarchy	14
Samples From a Fixed Dataset Using the UniformNonrationalBicubicBspline	15
Samples From a Fixed Dataset Using the UniformNonrationalBilinearBspline	15
Samples From a Fixed Dataset Using the StandardTriangleMesh	16
Samples From a Fixed Dataset Using the Bitmap	16
Central Server Approach	31
Collection of Group Servers Approach	31
Collection of Group Servers Approach	32
Fully Distributed Approach	32
The logical relationship between the Entity Service and the client applications on a single machine.	36
The logical communication channels between the Entity Service and the client applications.	38
A code fragment showing the state transfer interaction between the client application and the Entity Service.	40
A schematic representation of the state transfer interaction between the client application and the Entity Service.	40
A code fragment showing the miscellaneous transfer interaction between the client application and the Entity Service.	41
Booch Diagram of classes in the Entity Service	48
The logical relationship between the Terrain Service and the client applications on a single machine.	54
The logical communication channels between the Terrain Service and the client applications.	55
A code fragment showing the state transfer interaction between the client application and the Terrain Service.	57
A schematic representation of the "terrain changed" transfer interaction between the client application and the Terrain Service.	57
Booch diagram of classes for the Terrain Service	63
The logical relationship between the Fluid Service and the client applications on a single machine.	69
Logical communication channels between the Fluid Service and the client applications.	70
A code fragment showing the state transfer interaction between the client application and the Fluid Service.	72
A schematic representation of the "Fluid changed" transfer interaction between the client application and the Fluid Service.	72
Booch diagram of classes for the Fluid Service	77
Failure Plane	82
Soil mass division into n slices	83
Free body diagram for slice i	84
Analyzing the restoring force	85
Consideration of slices as containers	86

- An approximation of the 3-d configuration 88
- Booch diagram of soil model 91
- The staggered marker-and-cell mesh 100
- Relationships between pressure and height 102
- Fluid Source List and Tip List structure 105
- Fluid End List structure 106
- Fluid boundary structure 106
- Components of the fluidDTR 111
- Crater Profile and Parameters 114
- Booch diagram of crater model 117
- Dead-reckoning Anomaly 121
- Vehicle Instantaneous Track Points 123
- Track Profile and Parameters 123
- Terrain Service query relative to the vehicle 125
- Depression Component (Only) Used for Tracks 129
- Booch diagram for track model 130
- Minefield Two-tiered Approach 140
- Booch diagram of the Minefield DTR 142
- Visualizer control panel 148
- Booch diagram of Visualizer classes 149
- Simulation Design 151
- Classical View of Host-IG-UI 152
- Simulation Overview 158
- Class Diagram 160
- Current Design of Host-IG 164
- Host with an IG Server 165
- Best Design with Host-IG 166
- Design 1 167
- Design 2 167
- Design 3 168
- Design 4 169
- Design 5 169
- Design 6 170
- Design 7 171
- Design 1 173
- Design 2 174
- Design 3 175
- Design 4 176
- Design 5 Design 5 177
- Design 6 178
- Design 7 179
- Current Simulation Design 180
- Tracking Entities Through the IG Host Individually 181
- Tracking Entities Through the IG Host as Lists 182
- Tracking Entities Directly to the IG Individually 183
- Tracking Entities Directly to the IG as a List 184

Tracking DT Through the IG Host by CLOD 185
 Tracking DT Through the IG Host by Performer Lists 186
 Tracking DT Directly to the IG 187
 Dynamic Terrain Pipeline Version 1 188
 Complete Example of CLOD 189
 Stitching Together the CLOD Grids 190
 Creating Triangle Meshes from Grid 191
 Dynamic Terrain Pipeline Version 2 192
 Dynamic Terrain Pipeline Version 3 193
 Level 0 195
 Level 1 195
 197
 (a)“CLOD processing” Level 3 (b)“VP processing” Level 3 197
 “entity state processing” Level 3 198
 “IG” Level 3 199
 IG Host with a GUI 204
 Current Simulation Design 206
 GUI to Host Connection 207
 Sample GUI Class Diagram 210
 Sample Communication Interface Class Diagram 212
 GUI State and the GUI Service 213
perf2dtdb Main Window 233
 DTDB Coordinate System 235
perf2dtdb Data File Browser 236
 tscfg Window 243
 fscfg Window 250
 Module Booch class diagram 256
 DT Source Generator Main Window 257
 Entity Circle Window 258
 Detonations Window 260
 Fires Window 261
 Sinks GUI operating window 268
 Simulation Flow 276
 Schematics of the Breacher 277
 TPS Control Flow 278
 Vehicle System Schematics 279
 Power plant and torque converter characteristics 281
 Forces during a turn 283
 Blade Control Simulation Flow 285
 Blade Lift Mechanism 286
 Viable Mechanism Configuration 286
 Simulated blade mechanism configuration 287
 Tractive force versus speed curve 288
 Blade geometry 289
 Soil Attribute Layer 290
 Breacher clearing a mine field 291

Soil motion along digging tool 297
WES Blading Model 299
Projection of Digging Tool Facet onto a Plane 302
Front and side piling of soil 303
Wireframe model of breaching blade tool 307
Booch diagram of digging tool model 308

Acronyms

ADC - Automatic Depth Control
API - Application Programmer's Interface
AVLB - Armored Vehicle Launched Bridge
BV - BwanaVision
CFD - Computational Fluid Dynamics
CGF - Computer Generated Forces
CLOD, CTLOD - Continuous Terrain Level of Detail
CMV - Combat Mobility Vehicle
CPU - Central Processing Unit
CSG - Constructive Solid Geometry
DIS - Distributed Interactive Simulation
DFDB - Dynamic Fluid Database
DRM - Dead Reckoning Model
DT - Dynamic Terrain
DTDB - Dynamic Terrain Database
DTR - Dynamic Terrain Resource
DV - DracVision
ES - Entity Service
ESIG - Evans & Sutherland Image Generator
FS - Fluid Service
GL - Graphic's Language
GL - Graphics Library
GUI - Graphical User Interface
HAB - Heavy Assault Bridge
Hz - Hertz, frame rate, fields per second
IG - Image Generator
IP - Internet Protocol
IST - Institute of Simulation and Training (at the University of Central Florida, Orlando, FL)
I/ITSEC - Interservice/Industry Training Systems and Education Conference
MVC - Model View Controller
NTSC - Naval Training Systems Center
NURBS - Non-Uniform Rational B-Spline
OOP, OOPS - Object-Oriented Programming System
SGI - Silicon Graphics, Inc.
SIGGRAPH - Special Interest Group on Computer Graphics
STRICOM - Simulation Training and Instrumentation Command
TMS - Terrain Mapping Service
TPS - Terrain Positioning Service
TS - Terrain Service
TVS - Tracked Vehicle Simulator
UCF - University of Central Florida (in Orlando, FL)
UDP - User Datagram Protocol
UTM - Universal Transverse Mercator

UNRBS - Uniform Non-Rational B-Spline
VSL - Visual Systems Laboratory
WES - U.S. Army's Waterways Experiment Station

3D, 3-D - three dimensional

Dynamic Terrain Shared Environment

IST's research into dynamic environments can be characterized as the search for a substrate that supports dynamic, unscripted interactions within a distributed simulation. This includes not only higher-fidelity simulation models, but the interaction of high fidelity models with low fidelity models and when such interactions are reasonable. In short, the focus is on what must be done in order to support future simulation systems.

This research has led to a unifying semantic referred to as a Shared Environment. The Shared Environment concept represents a flexible, highly configurable representation of the state of the virtual environment. This approach seeks to define the interface between the state of the simulated world and the entities and functionalities that make up and affect that world.

The goals of the Shared Environment are that it must be scalable, vendor-independent, robust, and able to provide an easily understood interaction between players and the environment (i.e., a clean interface), support dynamic, unscripted changes to the environment (i.e., a flexible interface), and minimize interoperability issues (i.e., a consistent interface).

By defining simple semantics for the entire architecture, responsibilities of the components of that architecture can be clearly defined and the complexity of those components can be reduced. This is a benefit of an object oriented approach and is a major step towards a system that is more easily maintained and flexible enough to adapt to change.

The Shared Environment layer is responsible for providing information about the shared virtual space to the client applications, and for maintaining consistency of the representation of the shared virtual space. Some portion of the Shared Environment runs on each physical machine.

In this DT prototype, the Shared Environment is implemented as a set of services that provide information to the client applications. Information about conventional entities, as well as fires and detonations, is provided by a program called the Entity Service. Information about the attributes of the terrain, the logical terrain player, is provided by a program called the Terrain Service. In addition, a first cut implementation providing information about water flowing across the terrain is found in a program called the Fluid Service. The sections that follow describe each of these services in some detail.

1.0 Dynamic Terrain Database

1.1 Introduction to the DynamicTerrain Database

This document provides a detailed view of the active database element that is one component of the effort by the Visual Systems Laboratory at IST to design and prototype an architecture that will support the concept of dynamic terrain within a DIS simulation environment. The rest of the introduction will give some big picture information including goals for and assumptions about the active database element. Following that is a discussion about the abstract query concept, which is fundamental to understanding the database. The bulk of this document is contained in the sections that provide different views of the DynamicTerrainDatabase class. First is a conceptual, user's guide level view of the normal functioning of the class. Second is a detailed look at the internal architecture of the class from the standpoint of a client programmer.

1.1.1 Goals

Several design goals are related to the software class that implements the active database element. These include:

- decoupling the application from the form of the data - This is imperative with respect to the design of a flexible architecture. It is not safe to assume that the data forms currently in use will necessarily be used in the future, nor is it safe to assume that new desired functionalities will necessarily fit any currently used form.
- finding effective data abstractions for layered attributes - Decoupling the application from the form of the data requires effective data abstractions as intermediaries.
- supporting an arbitrary number of attributes - To provide for future demands, no artificial constraints should be placed on the number of attributes that can be used.
- flexibility - The active database element should be flexible with respect to how it handles attributes, queries and updates, and how it provides for future enhancements.

1.1.2 Assumptions about the Active Database Element

The active database element is by definition built within the current simulator context described above, so those assumptions generally apply. It is also necessary to enumerate and discuss the assumptions specific to the active database element. These include:

- a right-handed coordinate system with z up - The database uses a Cartesian coordinate system that is right-handed with the z-axis pointing up.
- loading the entire database - The current version has no paging mechanism for gracefully handling large databases. Image generator vendors have much experience in this area. Adding such functionality is not a research issue, and will not be dealt with here.
- extents of all attributes equal - Because each attribute to be loaded spans the same area, the extents for the database and all attributes will be equal.
- a constant response for query points outside of extents - If any individual x,y location

for which a query is made happens to fall outside of the extents, a constant value is returned. For attributes the value is zero. For surface normals, the value is -1.

- not handling feature data - The current version only handles layered data attributes.
- only one type of surface representation for updates - A configuration option, the type of surface representation to use for updates, doesn't change once set.
- global parameters are only used for texture coordinates - Currently, no external use for the parameters has been proposed except for texture coordinates. Control over these texture coordinate values is limited to a simple scaling.

Many of these restrictions represent refinements and not new ideas or functionalities. Also note that temporarily disregarding refinements and focusing on major functionalities is consistent with an object oriented approach. For example, the fact that the current version loads the entire database does not have a direct bearing on the design of the interface between the database class and the client application. Likewise, the fact that some subsequent version might provide a data paging mechanism to handle arbitrarily large databases is but a refinement and will not impact the interface between database and client. It is sufficient that the client application know that it can get data from the active database class. How the database class handles itself internally is not the client application's problem.

1.2 Analysis

Several areas have been identified as high-risk areas with respect to the active database element. These include:

- identifying a data abstraction for layered attributes - This is the means of transferring attribute data from the database to the user in a form that both makes sense and is decoupled from the actual underlying representation.
- developing and categorizing surface representations - In order to support a wide variety of attributes, it is necessary to develop a variety of surface representations that all know how to respond to the fundamental query. It is also necessary to try to establish some metrics for the representations so that users will have some help in determining how to configure any given scenario.
- merging of updated data - This is a traditional problem that is compounded by coupling the user application with the underlying representation.
- identifying a data abstraction for vectored attributes - This is the means of transferring vector data from the database to the user in a form that both makes sense and is decoupled from the actual underlying representation.
- update arbitration - This is the problem area that centers on two different requested changes for the same attribute in the same location at the same time.
- dead-reckoning of attributes - This is the problem of extrapolating changes to attributes (a generalization of the concept of dead-reckoning of an entity's position).

Of the six high-risk areas identified, the first three have been specifically addressed in this iteration of the database class. Some time has been spent considering each of the others, but the decision was made to push those into future iterations of the design. This is consistent with an object oriented approach, where it is important not to attempt too much in any one iteration.

1.2.1 The Abstract Form of the Query

The fundamental question to be asked of an active dynamic terrain database element is

"What is the value of attribute attr at location x,y?"

This question is thematic to our approach. It serves as the basis for all of the particular queries provided by the DynamicTerrainDatabase class. It is important to note that the query mechanism has been intentionally decoupled from the underlying representation. *Clients of the active database do not have, nor should they have, any preconceived notions about how the data are stored.* The original data may in fact have been an elevation grid, but from the standpoint of the user of the active database that piece of information is superfluous. This decoupling of the user from the data representation will prove even more useful as new types of data are incorporated into the system as a whole. It stands to reason that other desirable attributes (such as some forms of sensor data) may be better represented by something other than "standard" terrain structures (such as an elevation grid).

The client should be free to make requests for data at any point (within the extents). Given that, the database should be able to support queries that are arbitrarily spaced. This becomes a requirement of the underlying representation. The data representations must be able to answer a query at any point, not just at the specific points where they have a data value. For the purposes of understanding the DynamicTerrainDatabase class it is sufficient to assume that the underlying representations will return values anywhere within their extents.

Conversely, this fundamental question also serves as the basis for the update mechanism. That is, when a change to an attribute is desired, the question becomes a request to set the value of attribute attr at location x,y. Both the query and the update will be covered in more detail below.

1.3 Implementation

The implementation section of this document has two major sections. The first contains a user's guide level discussion of the DynamicTerrainDatabase. This includes the conceptual structure of the database, the area form of the query, configuring the DTDB, and the surface representations. The second section contains a client programmer's guide level discussion of the DynamicTerrainDatabase. This includes internal details of the class, as well as discussion of some of the source code.

1.3.1 User's Guide

This section introduces the DynamicTerrainDatabase class by taking a black-box approach. The functionalities of the class are observed from a conceptual standpoint to gain an understanding. In particular, not every detail will be covered here. For those who need finer detail, see the section below that focuses on the class from the client programmer's point of view.

The DynamicTerrainDatabase class described here is an attempt to provide an active database element required for future generations of simulators. The concept of an active database element represents a fundamental shift away from the static and towards the dynamic. The capabilities displayed by today's simulator databases only serve as an initial state for the kinds of simulators currently being dreamed of. *Simulator databases are in their infancy with respect to what they must become in order to support the levels of granularity and interaction desired of future systems.*

One major difference associated with the use of an active database element is the fact that the client will make frequent queries of the database to get fresh data as opposed to loading up the database once and flying around it. Recognition of this difference is a strong driving force behind the decoupling of the data representations from the client applications. Other important differences brought to light by an active database element are the need for an update arbitration mechanism to handle cases where two or more updates overlap and the possibility for dead reckoning of certain of the terrain attributes to reduce network bandwidth consumption. Some of these will be covered in more detail at a later time as the level of understanding grows.

1.3.1.1 The Conceptual Structure of the Database

It is important to visualize the conceptual structure of the database so as to better understand the query and update mechanisms. Remembering that at this point we are still living in a 2 1/2 D world, it is easy to think of the database as containing several planes of data, one for each attribute. See figure 1.

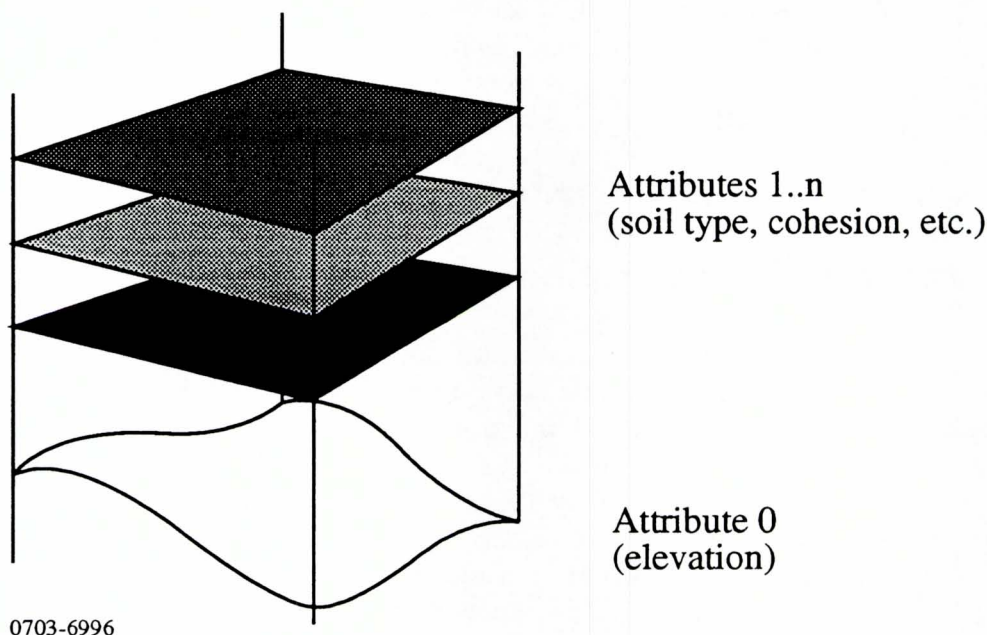


Figure 1. A Conceptual Model of the Data within the DynamicTerrainDatabase

For any x,y point within the extents of the database, there exists a vector (0..n-1) of information describing that point. This vector will contain elevation, as well as other attributes specified for that particular scenario. Thinking, then, in terms of the fundamental query mentioned above, get-

ting fresh data from the DynamicTerrainDatabase is a sequence of queries that span some 2D area and that span some number of attributes.

1.3.1.2 Segregation of Responsibility

Responsibilities are divided between the DynamicTerrainDatabase class and the client, or user, of the database class in a manner that reflects the principle goal of decoupling the client application from the data representations. The duties of the DynamicTerrainDatabase class include:

- maintaining representations for all necessary terrain-related attributes that can be both queried and updated by the class on behalf of the client application
- providing meaningful, yet generic forms for the transfer of data between the class and the client applications
- providing appropriate, useful, flexible mechanisms for the clients to query and update the data contained within the class
- providing configurability to the client, i.e., providing multiple representations for major functionalities of the database so that the client can choose the most appropriate representations based on his own situation

Responsibilities of the client include:

- properly instantiating the class, providing configuration and necessary data files
- allocating and maintaining the data buffers used to exchange data with the class
- correctly making use of the query and update mechanisms provided by the class (most importantly, not doing anything that depends on any particular internal knowledge that may be available about the DynamicTerrainDatabase class-- one important reason for decoupling the data representations from the client applications is so that the data representations can be changed if necessary without impacting the client application)
- translating the data into and out of the generic form provided by the class and any specialized internal form needed by the client application

1.3.1.3 Arbitrary Configuration

Arbitrary configuration is a desirable trait for the active database element. Only the data required need be loaded and worked with. This is also useful from the standpoint that each database element in the total system can be individually configured. An example is a networked scenario containing a stealth which is primarily concerned with elevation data and a vehicle simulator which is doing mobility calculations requiring not only elevation data, but also several other attributes of the soil. The database for the stealth need not load up other soil attributes, such as soil type, cone index, or soil strength, if they will not be used. In addition, it is also possible that the stealth might be able to run with coarser, or lower resolution, data. If the potential for configuration is truly arbitrary, then *scenario designers can evaluate their own needs and make their own decisions about what attributes are required and at what resolutions*, as opposed to technical-types making those decisions.

1.3.1.4 The Form of the Data Exchanged

Only layered type attributes are handled by the DynamicTerrainDatabase. Layered type attributes are wide area kinds of data such as elevation, soil type, or cohesion. Vector type attributes, such as roads and rivers are not maintained by the DynamicTerrainDatabase.

For the layered type attributes, the form for exchanging data between the DynamicTerrainDatabase and the user is an abstract data grid similar to an elevation grid. This basic data structure is simple enough to be easily manipulated, yet provides a fairly powerful representation for these attributes. In essence the abstract data grid is an m by n array of values sampled from the representation for the desired attribute. In practice, the abstract data grid has proven a good intermediary when referencing layered attributes such as elevation and soil type.

1.3.2 The Area Form of the Query

Practically speaking, when clients want fresh data they will most likely want it over some area. To this end, the actual query methods have been set up to handle the area form of the query. The area form allows the client to specify a minimal amount of information and can return a rather large amount of data. The area form is the query that makes use of the abstracted elevation grid.

The actual area form query looks like:

```
getAttributeValues(attr, q1, q2, q3, q4, m, n, buffer)
```

where:

`attr` is the attribute to be queried
`q1..q4` are 2D points that define the area
`m` and `n` are the number of samples in each direction
and `buffer` is the storage area into which data will be put

The client is responsible for determining the four 2D points in world coordinates that define the area of interest, determining the number of samples to be taken in each direction, and allocating the buffer to hold the data.

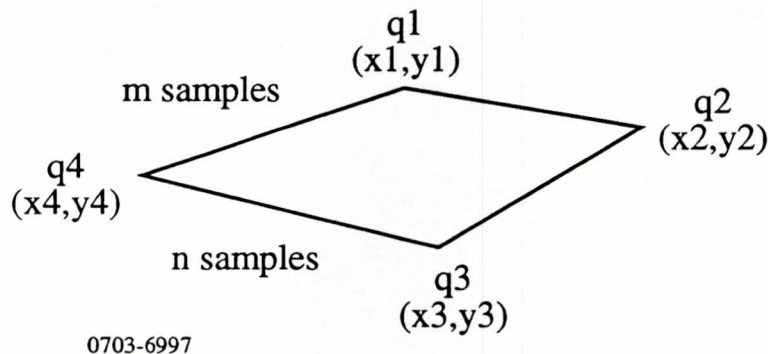


Figure 2. The Client's View of the Area Form of the Query

The $q1..q4$ are 2D coordinates in world space and represent an arbitrary quadrilateral area. From the standpoint of the DynamicTerrainDatabase, the corners do not have to represent an axis-aligned rectangle or even a rectangle at all. They can represent an arbitrary orientation, and they can degenerate into a triangle, a line segment, or even a single point. A single point query, then, is actually made by letting $q1=q2=q3=q4$ and $m=n=1$. Note that even though the query mechanism is flexible enough to allow for some pretty strange shapes, the added complexity that is associated with those strange shapes may not be something that the client program will want to deal with.

The DynamicTerrainDatabase will bilinearly interpolate between $q1..q4$ using m and n to find the locations of each individual x,y that will be used to query the underlying surface representation. The values returned by the underlying representation will be stored in the data buffer in row-major order where m represents the rows, and n represents the columns. Thus, the value at $q1$ is stored in `buffer[0]`, the value at $q2$ is stored in `buffer[n-1]`, the value at $q3$ is stored in `buffer[((m-1)*n) + (n-1)]`, and lastly the value at $q4$ is stored in `buffer[(m-1)*n]`. In general, running two loops as follows will walk across each row from left to right and from top to bottom:

```
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        printf("%d,%d    %f\n", i, j, buffer[i*n + j]);
```

1.3.2.1 Multiple Attribute Queries

Another practicality associated with queries is that clients who want values for several attributes will typically want them over the same area (values of $q1..q4$) and at the same resolution (values of m and n). To support this, a multiple attribute query has been set up that can be used to get values for an arbitrary number of attributes over the same area at the same resolution at the same time. More details on the multiple attribute query appear in the client programmer's guide section of this document.

1.3.2.2 Two Types of Clients

There are two general categories of clients for the DynamicTerrainDatabase. The passive client is one who does only queries for data--essentially a reader. The perfect example of the passive client is the image generator. The active client is one who both queries and updates data--a reader/writer. Examples of active clients are the resources, or models, that alter the elevation profile in response to some stimulus. More details on active and passive clients appear in the client programmer's guide section of this document.

1.3.2.3 Configuration

One important facet of the DynamicTerrainDatabase class is configurability. The client programmer has a significant amount of control over setting up the attributes he wants to deal with and how those attributes will be represented.

Below is a sample DynamicTerrainDatabase configuration file with extra comments. Several items are required to be present in the configuration file, and some are optional. Also, there are

certain dependencies with respect to the sequence of entries in the configuration file. These are spelled out below. Lines that begin with the '#' character are comments. Blank lines are ignored. The additional comments appear in *italics*.

The attribute keys define the mapping used by both the query and update mechanisms. If soil type is defined as attribute key 1, then the value for the attribute parameter in the query is 1. Note the convention of placing the elevation values in as attribute key 0. Hence, the value for the attribute parameter in any query for elevations is 0.

```
DTDB_CONFIG_FILE
#
Note that a valid configuration file begins with the character string
"DTDB_CONFIG_FILE". This string must be present or the DynamicTerrain-
Database will abort with an invalid configuration file message.
# DynamicTerrainDatabase configuration file
# =====
#
#     NOTE that the file must begin with the string DTDB_CONFIG_FILE!!
#
# -----
#
# several surface representations are currently available:
#
#     uniform_nonrational_bilinear_Bspline
#     uniform_nonrational_bicubic_Bspline
#     bitmap
#     standard_triangle_mesh
#
# anywhere you see surface_rep below, choose one of these
# -----
#
# the surface_rep to use for the temporary update surfaces
# the temporary surface rep defaults to
#     uniform_nonrational_bilinear_Bspline if not specified
#
The type of temporary surface representation specified here will be used for
all updates handled by the DynamicTerrainDatabase during a given scenario.

temporary_surface_rep  uniform_nonrational_bilinear_Bspline

# -----
#
# texture coordinate scaling values
# these values default to 1.0 if not specified
#
Setting these scale values larger than 1.0 will cause the texture coordinates
returned by the getAll queries to be scaled accordingly which usually has the
effect of tiling the texture applied to the terrain.

texture_scale_s      1.0
texture_scale_t      1.0
```

```

# -----
#
# extent quadrilateral in clockwise order
# extent quadrilateral values must be specified
#
This implementation requires that all attributes for a given scenario span the
same area, even though they can be at different resolutions.

p1_x      0.0
p1_y      0.0
p2_x      0.0
p2_y      4000.0
p3_x      4000.0
p3_y      4000.0
p4_x      4000.0
p4_y      0.0

# -----
#
# the number of attributes must be specified (including the
#     elevation as attribute 0) and must precede the attribute
#     definitions
#
This number includes the elevation as attribute 0, for example, if a given sce-
nario will use elevation, soil type, and cohesion then the number_of_attributes
would be 3.

number_of_attributes      3

# -----
#
# the main "surface" is elevation (the ground's geometry)
# this is a paired entry (data_file, surface_rep) that must
#     be specified (data_file before surface_rep)
# NOTE:  elevation is always attribute 0!!
#
Since elevation is by convention always attribute 0, it is not necessary to
specify the attribute key or the name.

elevation_data_file      sample.elevations.data
elevation_surface_rep    uniform_nonrational_bilinear_Bspline

# -----
#
# the following entries are 4-tuples (attribute_key,
#     attribute_name (with NO WHITESPACE!), data_file, surface_rep)
#     and represent additional attributes (beginning with key = 1)
# within the definition of an attribute, the attribute_key,
#     the attribute_name, and the data_file must precede
#     the surface_rep
# NOTE:  attribute_keys must be unique!!
#
The way that attribute keys work is directly linked to the way that a single
dimensional array is indexed. That is, if there are number_of_attributes

```

attributes in a given scenario, then the available attribute keys are in the range `[0..number_of_attributes-1]`. Remember that by convention elevation is always attribute 0, so any additional attributes will begin with key = 1, and increase by one until `number_of_attributes-1`. If these values are not consistent within the configuration file, or if any attribute definition doesn't contain all it's parts then the behaviour of the `DynamicTerrainDatabase` is undefined.

attribute_key	1
attribute_name	soil_type
data_file	sample.soiltype.data
surface_rep	bitmap
attribute_key	2
attribute_name	cohesion
data_file	sample.cohesion.data
surface_rep	standard_triangle_mesh

1.3.3 Programmer's Guide

This Programmer's Guide is broken into three major sections. The first takes a conceptual look at the internal structure of the `DynamicTerrainDatabase`. The second briefly discusses the surface representations used by the `DynamicTerrainDatabase`. And the third considers the details of the class, including the public definition from the header file, and sample client applications.

1.3.3.1 The Internal Structure of the Class

This section will take a look at the internal structure and workings of the `DynamicTerrainDatabase` class. An understanding of the material presented here is not at all required to use the class. This information does, however, provide for a deeper understanding of the functioning of the active database.

1.3.3.1.1 Segregation of Responsibility Within the Class

Again the goal of decoupling the application from the data representation serves as the principle for segregating responsibility, this time within the `DynamicTerrainDatabase` class. The duties of this class include:

- properly managing the appropriate surface representations as specified in the configuration file
- providing a query mechanism that translates an area query into a sequence of point queries, and passing those point queries on to the appropriate surface representation for processing
- providing an update mechanism that allows for a temporary surface representation corresponding to the updated area, and passing the temporary surface on to the appropriate surface representation for processing
- providing some means of generating reasonable texture coordinates for applications (such as an image generator) that require them

The responsibilities of the underlying surface representations include:

- being able to load from a data file or from a data set in memory
- being able to answer the point query for any x,y within their extents (flagging any query that falls outside the extents) and returning a vertex, a surface normal, or a pair of parameter values as required

1.3.3.1.2 The Internal Architecture

The internal architecture of the DynamicTerrainDatabase class is straightforward. The most important of the instance variables is the array of surface representations. There is one surface representation for each attribute in any given scenario. The DynamicTerrainDatabase manages the surface representations and provides the higher level logic associated with the queries and updates.

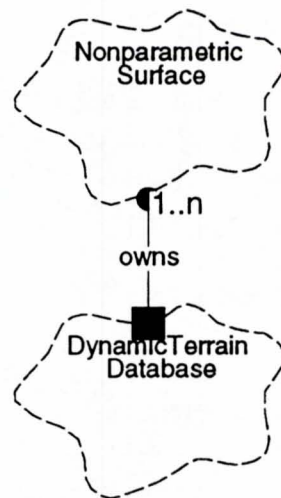


Figure 3. The DynamicTerrainDatabase (less simple instance variables)

As mentioned above, the DynamicTerrainDatabase class is responsible for translating the area form of the query into an appropriate sequence of point queries. This is done via simple bilinear interpolation between the corners of the query area and is covered in more detail below. Each area query then generates $m \times n$ point queries which are processed by the underlying surface representations. In other words, the database class processes the query up until the point that the actual calculations are done by the surface representation, and then the result from the surface representation is stored in the client's buffer by the database class.

Handling an update first involves instantiating a temporary surface representation with the updated data set. Then the temporary surface is passed on to the appropriate surface representation for processing. Each surface representation has the ability to update its data points based on another surface representation via a sequence of point queries much like the way the DynamicTerrainDatabase translates an area query into point queries.

1.3.3.1.3 Translating the Area Query into Point Queries

The DynamicTerrainDatabase class translates the area form of the query into a sequence of point queries by bilinearly interpolating between the four 2D corners of the query area. This is easiest to visualize in the axis-aligned rectangle case:

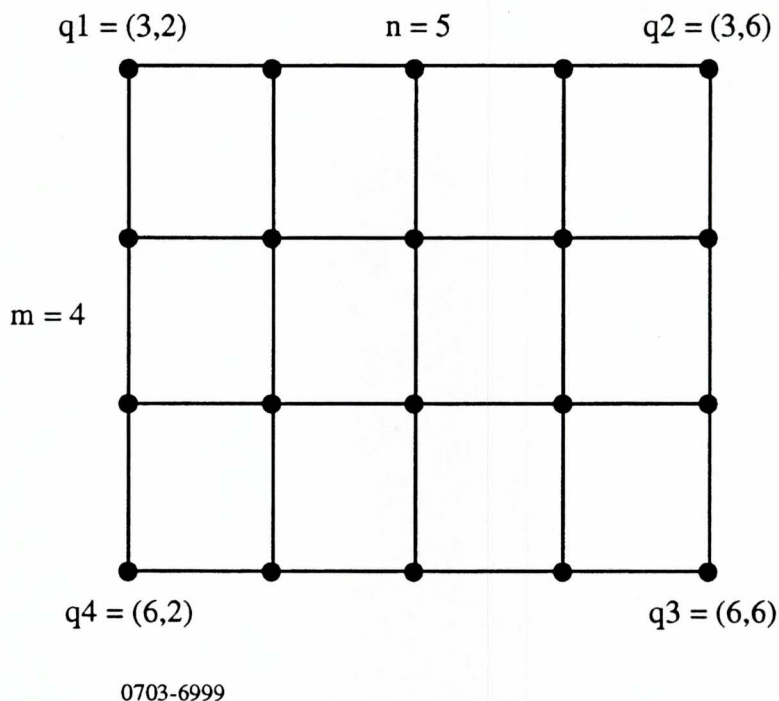


Figure 4. Translation of the Area Query into Point Queries

Each of the heavy dots in figure 4 above represents an x,y location that is derived by the DynamicTerrainDatabase class via bilinear interpolation between the corners of the query area based on the desired number of samples in each direction. Each of these locations becomes a point query to the appropriate underlying surface representation.

Recall that the query area is not required to represent an axis-aligned rectangle. The query area can be an arbitrarily oriented quadrilateral, and can degenerate into a triangle, a line segment, or even a point.

1.3.3.2 Surface Representations

An important part of decoupling the application from the underlying data representation is abstracting the interface between the two. In some sense, the DynamicTerrainDatabase class serves as an intermediary between the client and the data representation. The term surface representation is used *very generically* to refer to any of several 2 1/2 D representations that might be used for any particular attribute.

Currently, a class hierarchy of nonparametric surfaces has been created that includes bitmaps, standard triangle meshes, and uniform non-rational B splines of varying degree. Any other representation that might be deemed useful in representing some type of attribute can be added. *Note that within this hierarchy all of these siblings are functionally equivalent from the standpoint of the DynamicTerrainDatabase class.* That is to say, any attribute can be configured with any available surface representation without impact on the database class. The only differences between the siblings are the actual values returned and the computational cost incurred, because each sibling implements a different means of interpolation.

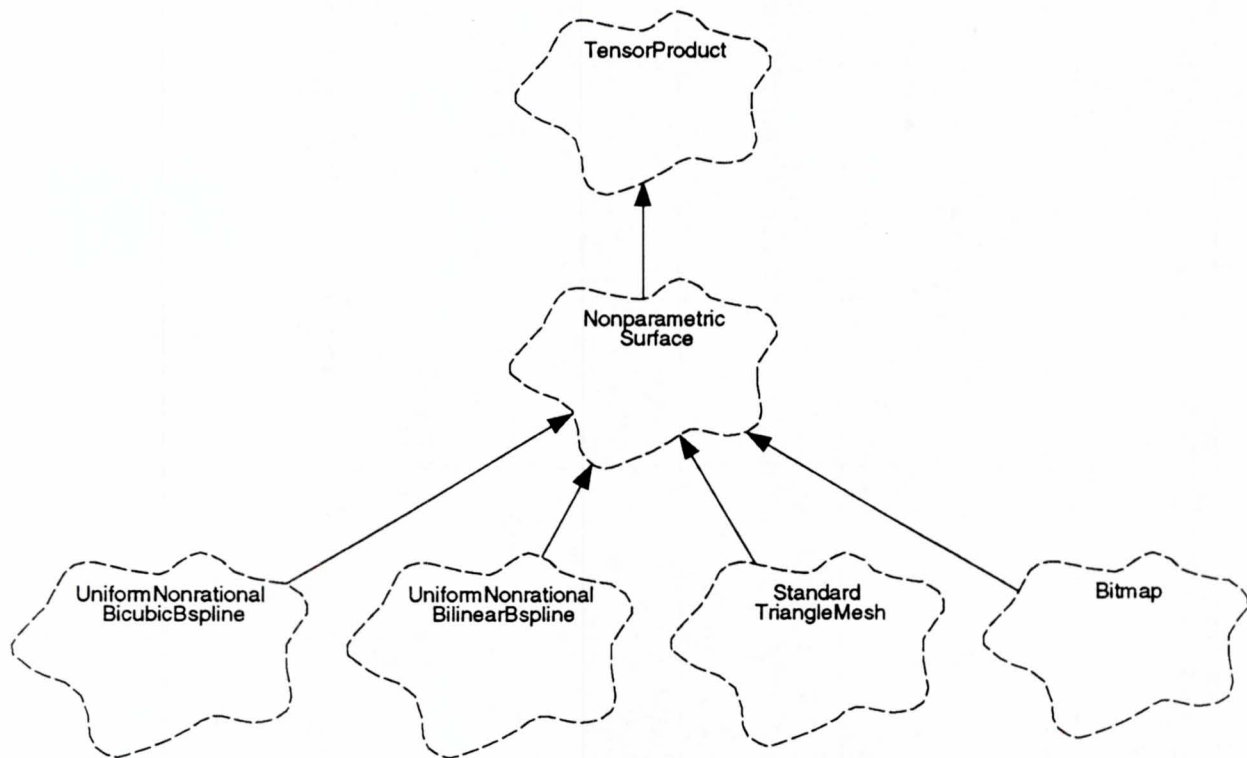


Figure 5. Surface Representations Class Hierarchy

Short descriptions of the surface classes follow, along with images of the different surface representations on the same underlying data set. Recall that what is important is the impact thinking about nonparametric surfaces had on the fundamental query, and that all of these surface representations are siblings in a class hierarchy making them interchangeable to the DynamicTerrainDatabase. The issue is which representation fits each attribute the best. The key is the flexibility to configure as appropriate.

1.3.3.2.1 The UniformNonrationalBicubicBspline Class

The UniformNonrationalBicubicBspline class implements a uniformly parametrized, nonrational, bicubic B-spline surface. It uses the deBoor algorithm at its core for finding points on the surface. See Figure 6 below. In practice, this class is used frequently as it gives a pleasant visual appear-

ance for many terrain attributes, even though it is slightly more computationally expensive than the bilinear.

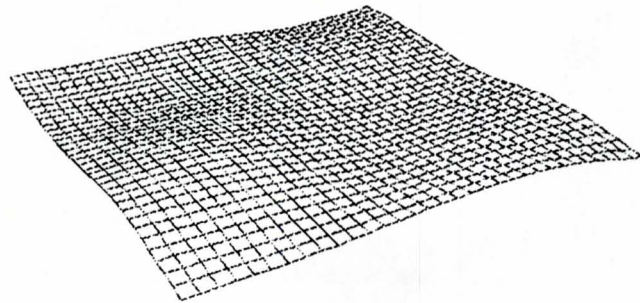


Figure 6. Samples From a Fixed Dataset Using the UniformNonrationalBicubicBspline

1.3.3.2.2 The UniformNonrationalBilinearBspline Class

The UniformNonrationalBilinearBspline class implements a uniformly parametrized, nonrational, bilinear B-spline surface. It also uses the deBoor algorithm at its core for finding points on the surface. See Figure 7 below. In practice, this class can be used for many terrain attributes as long as the spacing of the underlying data is not too large with respect to the sampling. As the sampling gets much denser than the underlying data then a "circus-tent" appearance emerges.

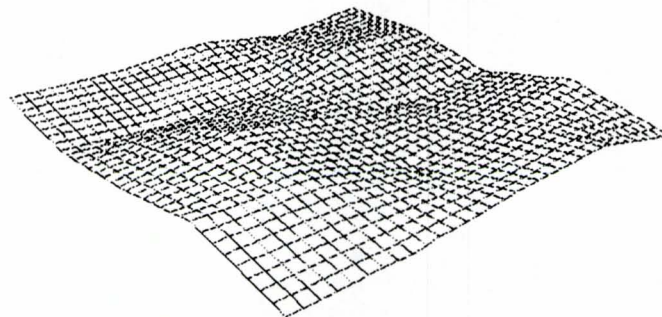


Figure 7. Samples From a Fixed Dataset Using the UniformNonrationalBilinearBspline

1.3.3.2.3 The StandardTriangleMesh Class

The StandardTriangleMesh class implements a regular triangulation breaking each set of 4 adjacent underlying data values into two triangles along the axis parallel to $y=x$. It uses plane equations at its core. See Figure 8 below. As the sampling gets much denser than the underlying data then the characteristic "big-polygon" appearance emerges.

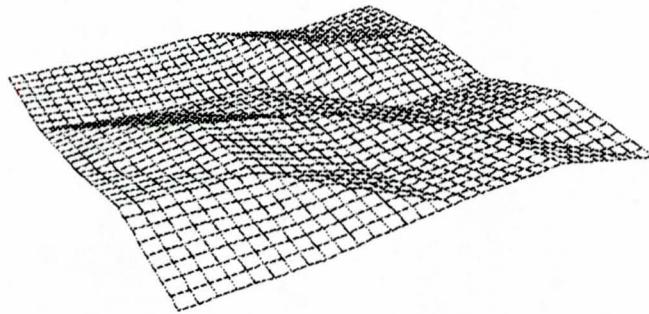


Figure 8. Samples From a Fixed Dataset Using the StandardTriangleMesh

1.3.3.2.4 The Bitmap Class

The Bitmap class implements a bitmap where any query between consecutive underlying data points returns a constant value equal to the data point closest to the parametric (0,0) corner of the database. See Figure 9 below. This class is not very useful to represent elevation, but certain other applications have been explored including marking distinct boundaries, and mapping images onto terrain.

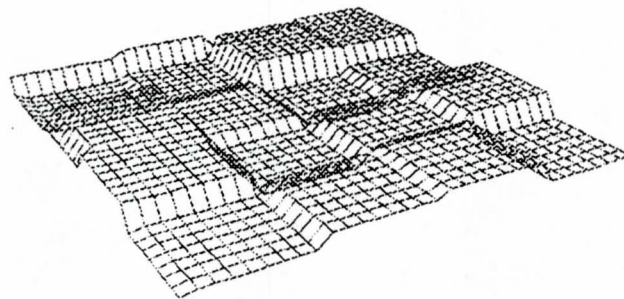


Figure 9. Samples From a Fixed Dataset Using the Bitmap

1.3.3.3 The DynamicTerrainDatabase Class

This section will focus on the details of how to incorporate the DynamicTerrainDatabase class into an application, that is to become a client of the database. The public portion of the class definition will be covered in detail. The two basic functionalities of the class, queries and updates, will be dealt with. The course of discussion here will assume some understanding of the DynamicTerrainDatabase class from previous sections. The configuration of the DynamicTerrainDatabase was covered previously.

There are three types of information that can be requested in a query. The first will be used most often and is the value of the attribute at the query point, in other words a point on the surface, or a vertex. The second type of information available is the surface normal at the query point. Usually, the only attribute normals requested will be for the elevation, but the query is sufficiently general that normals can be gotten for any attribute. The last type of information is essentially a pair of global parameters. Right now, these are only used as texture coordinates.

To reflect these three types, there are versions of the query method that get attribute values, and surface normals, as well as combinations that get values and normals. There are also versions that get values, normals and texture coordinates. Note that many of the methods are overloaded, with the only differences being the type of buffer. This overloading gives significant flexibility to the user of the DynamicTerrainDatabase in terms of getting data in the form he wants it.

1.3.3.3.1 The Class Definition

Following is the public portion of the class definition for the DynamicTerrainDatabase class as found in the DynamicTerrainDatabase.h++ header file augmented with more comments. Additional comments appear in italics. Note that the DynamicTerrainDatabase class, like several other classes used in the prototype, is subclassed off of ErrorNotification. This class provides simple, yet flexible error/status notification for the programmer. Since ErrorNotification has no direct impact on the DynamicTerrainDatabase, it will not be discussed further here. For more details, see ErrorNotification.h++.

```
class DynamicTerrainDatabase : public ErrorNotification
{
    public:
```

The constructor is called with the name of the configuration file to use, and the initial level of error notification. The constructor reads the configuration file, instantiates all of the necessary surface representations, and initializes the other instance variables.

```
    DynamicTerrainDatabase( char *config_filename, int initial_level
                          = INFO );
```

```
    ~DynamicTerrainDatabase();
```

```
//      if any individual point query fails, the returned value is:
//      0 or (0,0,0) for attributes
//      (0,0,-1) for normals
//      (0,0) for parameters
//
//      note the overloading on most query/update methods
//      the distinction is what the buffers look like
//      this provide significant flexibility for the user of the dtddb
//
//      single attribute query methods
//      -----
//
```

This method is the most basic area query, the SINGLE PLANE QUERY. The parameters represent the attribute from which to get values, the four corners of the query area, the number of samples

in each parametric direction, and a buffer to put the data values into. The method bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation, and stores the values in buffer.

```
int getAttributeValues( int attribute, float q1[], float q2[],
    float q3[], float q4[], int num_in_s, int num_in_t,
    float buffer[] );
```

This method is an area query that returns its results in three buffers, the TRIPLE PLANE QUERY. The parameters represent the attribute from which to get values, the four corners of the query area, the number of samples in each parametric direction, and three buffers to put the data values into. The method bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation, and stores the values in the buffers.

```
int getAttributeValues( int attribute, float q1[], float q2[],
    float q3[], float q4[], int num_in_s, int num_in_t,
    float vx[], float vy[], float vz[] );
```

This method is an area query that returns its results in one interlaced buffer, the TRIPLE PLANE INTERLACED QUERY. The parameters represent the attribute from which to get values, the four corners of the query area, the number of samples in each parametric direction, and an interlaced buffer to put the data values into. The method bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation, and stores the values in vertices.

```
int getAttributeValues( int attribute, float q1[], float q2[],
    float q3[], float q4[], int num_in_s, int num_in_t,
    float vertices[][3] );
```

This method is an area query that returns surface normals in three single planes, another TRIPLE PLANE QUERY. The parameters represent the attribute from which to get normals, the four corners of the query area, the number of samples in each parametric direction, and three buffers to put the components of the normals into. The method bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation, and stores the values in the buffers.

```
int getAttributeNormals( int attribute, float q1[], float q2[],
    float q3[], float q4[], int num_in_s, int num_in_t,
    float nx[], float ny[], float nz[] );
```

This method is an area query that returns surface normals in one interlaced buffer, another TRIPLE PLANE INTERLACED QUERY. The parameters represent the attribute from which to get normals, the four corners of the query area, the number of samples in each parametric direction, and an interlaced buffer to put the components of the normals into. The method bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation, and stores the values in normals.

```
int getAttributeNormals( int attribute, float q1[], float q2[],
    float q3[], float q4[], int num_in_s, int num_in_t,
    float normals[][3] );
```


This method is an area query that combines the triple plane value and surface normal queries. The parameters represent the attribute from which to get values, the four corners of the query area, the number of samples in each parametric direction, three buffers to put data values into, and three buffers to put the components of the surface normals into. The class bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation, and stores the values in the buffers.

```
int getAttributeValuesAndNormals( int attribute, float q1[],
    float q2[], float q3[], float q4[], int num_in_s,
    int num_in_t, float vx[], float vy[], float vz[],
    float nx[], float ny[], float nz[] );
```

This method is an area query that combines the triple plane interlaced value and surface normal queries. The parameters represent the attribute from which to get values, the four corners of the query area, the number of samples in each parametric direction, an interlaced buffer to put data values into, and an interlaced buffer to put the components of the surface normals into. The method bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation, and stores the values in the buffers.

```
int getAttributeValuesAndNormals( int attribute, float q1[],
    float q2[], float q3[], float q4[], int num_in_s,
    int num_in_t, float vertices[][3], float normals[][3] );
```

This method is an area query that combines the triple plane value and surface normal queries and also returns texture coordinates, in other words all the planes of data possible from the dtddb. The parameters represent the attribute from which to get values, the four corners of the query area, the number of samples in each parametric direction, three buffers to put data values into, three to put the components of the normals into, and two to put texture coordinates into. The method bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation, and stores the values in the buffers.

```
int getAll( int attribute, float q1[], float q2[], float q3[],
    float q4[], int num_in_s, int num_in_t, float vx[],
    float vy[], float vz[], float nx[], float ny[],
    float nz[], float tu[], float tv[] );
```

This method is an area query that combines the triple plane interlaced value and surface normal queries and also returns texture coordinates, in other words all the planes of data possible from the dtddb. The parameters represent the attribute from which to get values, the four corners of the query area, the number of samples in each direction, an interlaced buffer to put data values into, one to put the components of the normals into, and one to put texture coordinates into. The method bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation, and stores the values in the buffers.

```
int getAll( int attribute, float q1[], float q2[], float q3[],
    float q4[], int num_in_s, int num_in_t,
    float vertices[][3], float normals[][3],
    float textures[][2] );
```

```

//
//      multiple attribute query method
//      -----
//
//      ----- works for single plane queries only!! -----
//
//      num_attributes is how many attributes are in this query
//      attributes is an array of integers each of which is an attribute key
//      it is assumed that there are num_attributes keys in attributes[]
//      buffers is an array of pointers to buffers (each is a single plane)
//      it is assumed that there are num_attributes pointers in buffers[]
//

```

This method is the single plane multiple attribute query, a SINGLE PLANE QUERY for each of several attributes over the same area, at the same spacing, at the same time. The parameters represent how many attributes are in this query, the attribute keys, the four corners of the query area, the number of samples in each direction, and buffers to put the data values into. The method bilinearly interpolates between the q1..q4 to determine all of the individual point queries, queries the underlying surface representation for each attribute, and stores the values in the buffers.

```

int getMultipleAttributeValues( int num_attributes, int *attributes,
                                float q1[], float q2[], float q3[], float q4[],
                                int num_in_s, int num_in_t, float *buffers[] );

```

```

//
//      single attribute update methods
//      -----
//

```

This method is the most basic area update, the SINGLE PLANE UPDATE. The parameters represent the attribute to update, the four corners of the update area, the number of samples in each parametric direction, and a buffer that holds the new data values. The method creates a temporary surface representation that covers q1..q4 and uses the new data, and then hands the temporary surface over to the attribute for update.

```

int handleUpdate( int attribute, float q1[], float q2[], float q3[],
                  float q4[], int num_in_s, int num_in_t, float buffer[] );

```

This method is the triple plane area update, the TRIPLE PLANE UPDATE. The parameters represent the attribute to update, the four corners of the update area, the number of samples in each parametric direction, and three buffers that hold the new data values. The method creates a temporary surface representation that covers q1..q4 and uses the new data, and then hands the temporary surface over to the attribute for update.

```

int handleUpdate( int attribute, float q1[], float q2[], float q3[],
                  float q4[], int num_in_s, int num_in_t, float x[],
                  float y[], float z[] );

```

```

//
//      multiple attribute update method
//      -----
//
//      ----- works for single plane updates only!! -----
//
//      corresponds to multiple attribute query above
//

```


This method is the single plane multiple attribute update, a SINGLE PLANE UPDATE for each of several attributes over the same area, at the same spacing, at the same time. The parameters represent how many attributes are in this update, the attribute keys, the four corners of the update area, the number of samples in each parametric direction, and buffers that hold the new data values. The method creates a temporary surface representation that covers q1..q4 and uses the new data for each attribute, and then hands the temporary surface over to the attribute for update.

```

        int handleMultipleUpdates( int num_attributes, int *the_attributes,
                                   float q1[], float q2[], float q3[], float q4[],
                                   int num_in_s, int num_in_t, float *buffers[] );

//
//      miscellaneous methods
//      -----

//      set values for texture scaling
//
void setTextureScaling( float tex_scale_s, float tex_scale_t );

//      get extents of an attribute (generally the same for all attributes)
//
int getExtents( int attribute, float q1[], float q2[], float q3[],
               float q4[] );

//      dumps current state of dtddb (generally for debug purposes)
//
void dumpState( FILE *outfile );

//      write an attribute out to a file
//      'filename' is the name of the new dtddb file
//      'attribute' is the attribute to write out
//      'z_values_only' is a flag, when
//          FALSE - x, y, and z values written into file
//          TRUE  - only z values written into file
//
void writeDTDBFile( char *filename, int attribute, int z_values_only );
};

#endif

```

1.3.3.3.2 The Passive Client

As mentioned above, the passive client is one who is only concerned with making queries of the DynamicTerrainDatabase class. To aid in understanding how to interact with the database, a sample program is provided. This sample program is also available as passiveClient.c++ in the release directory. In the listing below, items and actions related to the DynamicTerrainDatabase are shown in boldface type.


```

////////////////////////////////////
//
// FILENAME:
//         passiveClient.c++
//
// Description:
//         simple test program demonstrating interaction
//         between passive client and DynamicTerrainDatabase
//
// By      - Visual Systems Laboratory -
//         - Institute for Simulation and Training -
//         - University of Central Florida -
//
// Copyright (c) 1995 the University of Central Florida.
//         - All Rights Reserved.
//
// Author(s):
//         Marty Altman
//
// Hardware:
//         Written on Silicon Graphics under IRIX 5.3, and
//         uses GL.
//
// Warning(s):
//
// Note(s):
//         note that certain auxiliary declarations and functions
//         (mostly GL stuff) are located in aux.h++ and aux.c++
//         these are not integral to the interaction of the client
//         with the dtdb, but do provide a simple example of how
//         to draw pictures with the data you get back
//
// Update(s):
//         June 21, 1995    release version
//
////////////////////////////////////

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#include "aux.h++"
#include "DynamicTerrainDatabase.h++"

// max number of data points in one dimension (the buffers are
//         DATA_SIZE x DATA_SIZE points)
#define DATA_SIZE    257

// default name of configuration file
#define NAME          "dtdb.cfg"

// buffers for the data (to be filled by the dtdb)
float  points[DATA_SIZE*DATA_SIZE][3],

```

```

        normals[DATA_SIZE*DATA_SIZE][3];

// the active database
DynamicTerrainDatabase *dtdb;

// declared here, and potentially used and/or changed elsewhere...
// (e.g., the user_interaction function in aux.c++)
int      ns = 9,          // number of samples in
        nt = 9,          // each direction
        resample;        // flag for resampling the database

float    ll[2], lr[2], ur[2], ul[2];    // extents of query area

void get_data( void ) {
// get extents of attribute 0 (elevation) (in a real program, could
// be buffered instead of making the query each time) (or, more
// likely, a real program will determine its own query extents
// because they will be smaller than the extents of the database)
    dtdb->getExtents(DTDB_ELEVATION, ul, ur, lr, ll);

// get data (values and normals) for attribute 0 (elevation)
    dtdb->getAttributeValuesAndNormals(DTDB_ELEVATION,
        ul, ur, lr, ll, ns, nt, points, normals);
}

void main( int argc, char *argv[] ) {
    char    filename[100];
    int     i, foreground;

// allow for override of default values via command line
    strcpy(filename, NAME);
    printf("defaulting filename to [%s]\n", filename);
    foreground = TRUE;
    printf("defaulting foreground TRUE\n");

    for (i=1; i<argc; i++) {
        if ((!strncasecmp(argv[i], "-h", 2)) ||
            (!strncasecmp(argv[i], "?", 1))) {
            printf("\nusage:\n\t%s [-h] [-f] [-c<configfile>]\n", argv[0]);
            exit(0);
        }
        if (!strncasecmp(argv[i], "-nf", 3)) {
            foreground = FALSE;
            printf("setting foreground FALSE\n");
        }
        if (!strncasecmp(argv[i], "-c", 2)) {
            strcpy(filename, &(argv[i][2]));
            printf("setting filename to [%s]\n", filename);
        }
    }

// instantiate the dtdb
    dtdb = new DynamicTerrainDatabase("test dtdb", filename);

```

```

// initialize graphics
initialize_graphics( foreground );

// perform query (to get data)
get_data();
resample = FALSE;

// -----
// main loop...
//
while (TRUE) {

    // if user interaction caused need to resample, do so
    if (resample) {
        get_data();
        resample = FALSE;
    }

    // begin a new frame
    graphics_begin_frame();

    // draw the elevation data
    draw_it(ns, nt, points, normals);

    // end the frame
    graphics_end_frame();

    // check for / handle user interaction
    user_interaction();

}
}

```

1.3.3.3.3 The Active Client

Once the interaction for a passive client is understood, the additional interaction needed for an active client is straightforward. Basically, a query is done over the desired area then changes are made to the data in the buffer based on whatever model the active client represents. Finally, the updated buffer is handed back to the database via a `handleUpdate` method. The responsibility for actually merging the data resides within the `DynamicTerrainDatabase`. Note that the query mechanism is the same for an active client of the database as it was for the passive.

To aid in understanding how to interact with the database, a sample program is provided. This sample program is also available as `activeClient.c++` in the release directory. In the listing below, items and actions related to the `DynamicTerrainDatabase` are shown in boldface type.


```

////////////////////////////////////
//
// FILENAME:
//         activeClient.c++
//
// Description:
//         simple test program demonstrating interactions
//         between active client and DynamicTerrainDatabase
//
// By      - Visual Systems Laboratory -
//         - Institute for Simulation and Training -
//         - University of Central Florida -
//
// Copyright (c) 1995 the University of Central Florida.
//         - All Rights Reserved.
//
// Author(s):
//         Marty Altman
//
// Hardware:
//         Written on Silicon Graphics under IRIX 5.3, and
//         uses GL.
//
// Warning(s):
//
// Note(s):
//         note that certain auxiliary declarations and functions
//         (mostly GL stuff) are located in aux.h++ and aux.c++
//         these are not integral to the interaction of the client
//         with the dtddb, but do provide a simple example of how
//         to draw pictures with the data you get back
//
// Update(s):
//         June 21, 1995    release version
//
////////////////////////////////////

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#include "aux.h++"
#include "DynamicTerrainDatabase.h++"

// max number of data points in one dimension (the buffers are
//     DATA_SIZE x DATA_SIZE points)
#define DATA_SIZE    257

// default name of configuration file
#define NAME          "dtddb.cfg"

```

```

// buffers for the data (to be filled by the dtddb)
float  points[DATA_SIZE*DATA_SIZE][3],
        normals[DATA_SIZE*DATA_SIZE][3],
        update[DATA_SIZE*DATA_SIZE];

// the active database
DynamicTerrainDatabase  *dtddb;

// declared here, and potentially used and/or changed elsewhere...
//      (e.g., the user_interaction function in aux.c++)
int      ns = 9,          // number of samples in
        nt = 9,          //      each direction
        update,          // flag for updating the database
        resample;        // flag for resampling the database

float  ll[2], lr[2], ur[2], ul[2];    // extents of query area


void get_data( void ) {
//
// get extents of attribute 0 (elevation) (in a real program, could
//      be buffered instead of making the query each time) (or, more
//      likely, a real program will determine its own query extents
//      because they will be smaller than the extents of the database)
    dtddb->getExtents(DTDB_ELEVATION, ul, ur, lr, ll);

// get data (values and normals) for attribute 0 (elevation)
    dtddb->getAttributeValuesAndNormals(DTDB_ELEVATION,
        ul, ur, lr, ll, ns, nt, points, normals);
}


void update_data( void ) {
    int i,j;

// make an arbitrary change to the values:
//      add a value to each data point that is between -5 and 5
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            update[i*n+j][2] = points[i*n+j][2] +
                (((float)rand() / 65535.0) * 2.0) - 1.0) * 5.0);
//          (----between 0 and 1----)
//          (-----between 0 and 2-----)
//          (-----between -1 and 1-----)
//          (-----between -5 and 5-----)

// update data (values and normals) for attribute 0 (elevation)
    dtddb->handleUpdate(DTDB_ELEVATION, ul, ur, lr, ll, ns, nt, update);
}

```

```

void main( int argc, char *argv[] ) {
    char    filename[100];
    int     i, foreground;

    // allow for override of default values via command line
    strcpy(filename, NAME);
    printf("defaulting filename to [%s]\n", filename);
    foreground = TRUE;
    printf("defaulting foreground TRUE\n");

    for (i=1; i<argc; i++) {
        if ((!strncasecmp(argv[i], "-h", 2)) ||
            (!strncasecmp(argv[i], "?", 1))) {
            printf("\nusage:\n\t%s [-h] [-f] [-c<configfile>]\n", argv[0]);
            exit(0);
        }
        if (!strncasecmp(argv[i], "-nf", 3)) {
            foreground = FALSE;
            printf("setting foreground FALSE\n");
        }
        if (!strncasecmp(argv[i], "-c", 2)) {
            strcpy(filename, &(argv[i][2]));
            printf("setting filename to [%s]\n", filename);
        }
    }

    // seed the random number generator
    srand(1234);

    // instantiate the dtddb
    dtddb = new DynamicTerrainDatabase("test dtddb", filename);

    // initialize graphics
    initialize_graphics( foreground );

    // perform query (to get data)
    get_data();
    update = FALSE;
    resample = FALSE;

    // -----
    // main loop...
    //
    while (TRUE) {

        // if user interaction caused need to update, do so
        if (update) {
            update_data();
            update = FALSE;
        }
    }
}

```



```

// if user interaction caused need to resample, do so
if (resample) {
    get_data();
    resample = FALSE;
}

// begin a new frame
graphics_begin_frame();

// draw the elevation data
draw_it(ns, nt, points, normals);

// end the frame
graphics_end_frame();

// check for / handle user interaction
user_interaction();

}
}

```

2.0 Abstract Service

2.1 Introduction to the Abstract Service

This document describes some of the aspects of the Abstract Service which is the name applied to code that has been abstracted between all Services. It combines all common functionality of the Services together into one place for ease of maintenance, iterative design, etc. For example, the basic control structures of all the Services are the same so the structures and access methods have all been abstracted into one location. The remainder of this document details some of the more important pieces that have been abstracted as well as the reasoning behind the decision.

2.2 Analysis

When looking at the abstract pieces of a service, two types are seen: pieces of the service itself as well as pieces that are used by the service. For the services built under the Dynamic Terrain (DT) project at the Institute for Simulation and Training (IST), three pieces have been implemented that are part of each Service (see the Entity Service, Terrain Service and Fluid Service documents) and two pieces have been implemented that can be used by each service. The following subsections talk about what each piece is and the reasoning behind them.

2.2.1 Service

The Service class contains the code for the routing of packets in different network configurations. Currently, a service can run in four different modes: server, client, relay and distributed. They are described as follows:

- *server*. This is meant for a service that is running as a central server. This means that it merges updates with its database *immediately* and forwards the update to *all* network links.
- *client*. This is for the clients of a server. It does *not* merge client updates immediately but does forward it to all *other* network links (expecting it to return when "approved" by a server).
- *relay*. This is a relay between servers and clients. It does not merge immediately but forwards the update "up" to the server. Note that it must know which link is the one going up to the server (see the Terrain Service and Fluid Service documents).
- *distributed*. This is the mode most often used in the IST DT Project for the shared environment concept. It merges client updates immediately and forwards it to the network links.

This allows the analysis of different network configurations as a possible Distributed Interactive Simulation (DIS) study. The Appendix shows diagrams of four possible configurations using these modes.

2.2.2 Service Interface

The ServiceInterface class deals with the allocation of the shared memory for the control channel (including the access tables for each client). It also includes code for locking and unlocking these channels using the semaphore structure. Note that this class can be replaced in the future as different communication structures (between service and client) are designed.

2.2.3 Client Interface

The ClientInterface class performs the same functions as the ServiceInterface class, but does it on the client side of the communication. It attaches to the shared memory (rather than allocating it also) as well as locking and unlocking the control semaphores.

2.2.4 Networking

Obviously, networking is used by all services. A small hierarchy has been developed that includes both unicast (point-to-point) and broadcast protocols. The parent class deals with the actual opening of a socket, and contains the reading and writing methods. The individual subclasses perform the actual binding of the port.

2.2.5 Coordinate Conversion

The coordinate conversion class is also something that can be used by all classes. Currently, geocentric, flat earth, ESIG and Performer coordinates are supported (see the Coordinate Conversion appendix). Note that only the Entity Service uses this coordinate conversion class at this time.

2.3 Implementation

As more aspects of the services can be abstracted, the more complex these core classes will become and the easier the maintenance of them will be. For example, it may be possible to abstract the general concept of processing a PDU (and updating a database) into the Abstract Service. In addition, it may be possible to generalize the configuration files of the services into one configuration file that can be read by the Abstract Service. Obviously, the more code that can be made common between the services, the easier maintenance will become.

2.4 Appendix

This appendix contains four possible configurations of the network modes supported by the Service class. Note that only the Terrain Service and Fluid Service use these (since the Entity Service does broadcast).

2.4.1 One Central Server

- All changes go to one location as a **request**

- Change is **not** automatically merged into local database
- Central Server sends update back as just another network terrain change

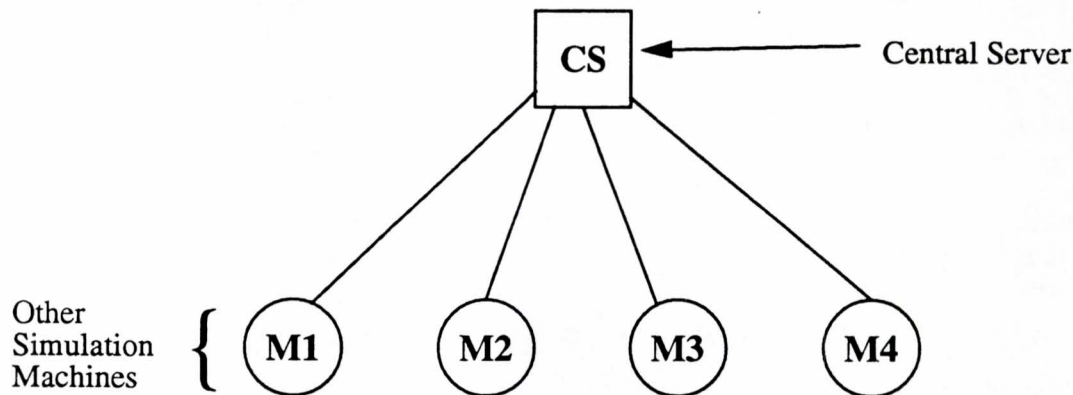


Figure 10. Central Server Approach

2.4.2 A Collection of Group Servers

- All changes go to one **local** location as a request
- Change is **not** automatically merged into local database
- Group Server **does** merge it in automatically and sends it back to local machines
- Group Server sends it on to other Group Servers

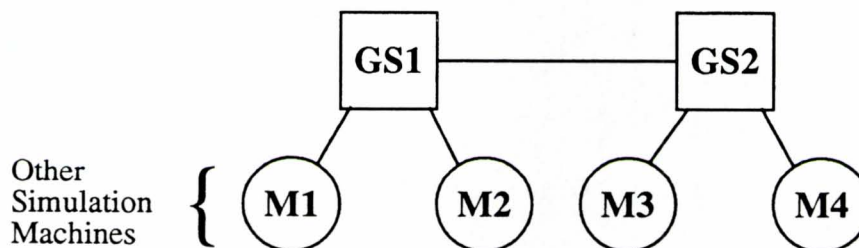


Figure 11. Collection of Group Servers Approach

2.4.3 One Central Server with Group Relay Servers

- All changes go to one **local** location as a request
- Change is **not** automatically merged into local database
- Group Server does **not** merge it in automatically
- Group Server sends it onto other group servers or the one central server (acting as a relay)

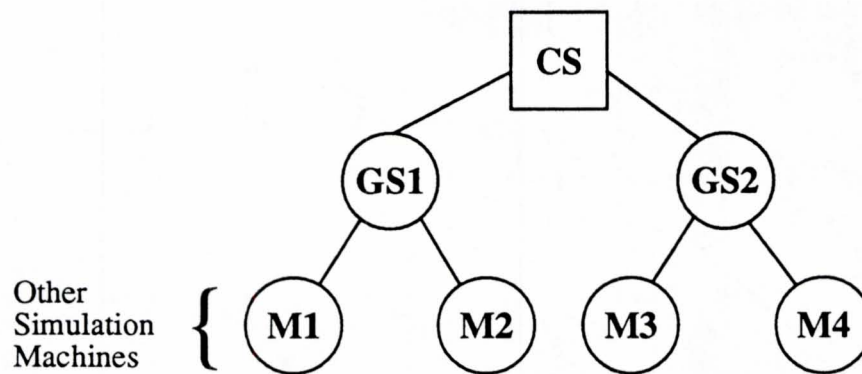


Figure 12. Collection of Group Servers Approach

2.4.4 Fully Distributed

- All changes go to all locations as an update
- Change is automatically merged into local database immediately

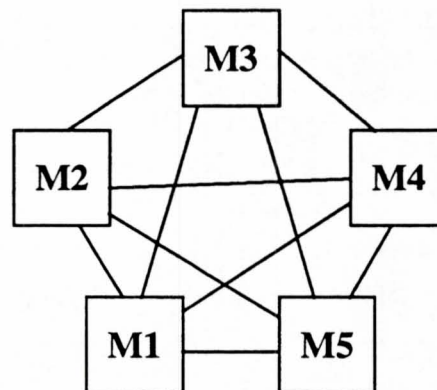


Figure 13. Fully Distributed Approach

3.0 Entity Service

3.1 Introduction to Entity Service

One of the requirements of the DIS environment is that each node maintain representations for other simulated entities (i.e., ghosts or remote entity approximations). Entity State PDUs are read from the DIS network for the entities, and dead reckoning is performed on the remote entity approximations. In an environment where there is only one application per physical machine it matters little where this entity state functionality resides. On the other hand, if the capability to have several applications reside on one physical machine exists, the location of this functionality becomes important.

3.1.1 Background

At a high level, Entity State PDU processing and dead reckoning address one issue of sharing the environment. In distributed interactive simulations, it is necessary for each entity to have a logical view of each of the other entities, and that all of the representations for a given entity are the same within a specified tolerance. One component of a shared environment is the current state of all the entities within that environment. Other components of a shared environment are discussed in another section of this documentation.

Consider the capability of having several applications resident on one physical machine. From the pragmatic standpoint, many of the functionalities associated with the Dynamic Terrain effort (e.g., cratering, soil slumping) are individually relatively small. In particular, many of them are too small to warrant allocation of a machine. This leads to two fundamental approaches. Many, perhaps all, of these small Dynamic Terrain functionalities could be bundled into one large eclectic application, or each of these small functionalities could reside in separate applications.

If the Dynamic Terrain capabilities are bundled into one large application, the question of where to place the entity state functionality is easily answered, but the ramifications upon software complexity are significant. Both development and maintenance are unnecessarily complicated by the increased coupling and weaker encapsulation of the Dynamic Terrain functionalities.

If the Dynamic Terrain capabilities are implemented through several small applications, the software issues are mediated if a standard application interface is assumed. However, placement of the entity state functionality becomes a problem. Duplicating this functionality is not only wasteful, but causes difficulty in managing the physical communication of PDUs.

It is important that a shared environment exist, and it would be ideal if applications were insulated from the duties of managing that shared environment. Better encapsulation of the environment, and the applications, not only simplifies development, but also helps to localize the impact of adding new capabilities in the future. Whether a dynamic terrain application or a more "standard" simulator, the software can be dramatically easier to build and maintain with prudent use of abstraction and encapsulation.

3.1.2 Problem Statement (external)

The requirement for the Entity Service is a single application that performs DIS I/O (input/output), dead reckons remote entity approximations, and supports simultaneous client applications that require independent subsets of entity state information.

3.1.3 Solution

Several design goals are related to the Entity Service, including:

- Decouple the application from the form of the data - This is imperative with respect to the design of a flexible architecture. It is not safe to assume that the data forms currently in use will necessarily be used in the future, nor is it safe to assume that new desired functionalities will necessarily fit any currently used form. In this case, the form referred to is the communications protocol (DIS PDUs). Further, this keeps our architecture independent of any particular vendor's implementation, as well as independent of any particular type of simulation (virtual, constructive, CGF).
- Find effective data abstractions for relaying entity information - Decoupling the application from the form of the data requires the development of effective data abstractions that can be used as intermediaries.
- Support a small, but arbitrary number of applications - To provide for future demands, no artificial constraints should be placed on the number of client applications that can be running on a single machine at any given time. The principal constraints should be performance requirements and resource availability. If twenty lightweight applications could perform on a single machine, it should be possible to allocate all twenty of those to a single machine.
- Flexibility - The Entity Service should be flexible enough to handle changes to the protocols (both between the service and the network, and between the service and the applications). Future requirements, such as mobility and Newtonian physics, may require additional state information for each entity. A sandbox for testing experimental protocol changes is required.
- Perform dead reckoning - Dead reckoning for entities must be performed once per physical machine, but it doesn't need to be done once per application. Dead reckon in one place on the machine and make that information available to all applications on the machine.

3.2 Analysis

The cornerstone of our approach is the use of object-oriented analysis and design techniques. The object-oriented paradigm is characterized by an iterative, incremental approach in which both the understanding of the problem and the design of a solution for that problem evolve over time from the general towards the specific. It is important to get results early, and to use the knowledge gained to refine subsequent iterations.

As part of our iterative approach, we have chosen to enumerate and address high-risk areas as early as possible. In the early iterations, we wanted to face the issues that had the highest potential to cause radical changes in the design. We wanted to build a reasonable skeleton before we tried to flesh it out. Areas identified as high-risk for the Entity Service include:

- Coordinate conversions - Where should these take place? What coordinate systems will be supported? Who should have responsibility for getting into and out of different coordinate frames?
- Identifying a data abstraction for entity state information - This abstraction ultimately becomes the sandbox for experimental protocols. What are possible/reasonable/useful ways to handle this experimentation?
- Interprocess communications - What are possible/reasonable/useful ways to handle communication between the service and the client applications? What are the tradeoffs between performance and complexity?

3.3 Implementation

3.3.1 Problem Statement

Taking into account the stated design goals and the desire to address the high-risk areas early, a single application is needed to serve as the interface between the client applications and the shared environment.

3.3.2 User's Guide

3.3.2.1 The Structure of the Relationship

The Entity Service runs in its own process and serves as the intermediary between the client applications and the DIS network. There is one copy of the Entity Service per physical machine. See figure below.

We encapsulated the functionalities of DIS communication and dead reckoning within the Entity Service. Details about the dead reckoning implementation are found in the <<Dead Reckoning Appendix document>>. We have also gained the capability to support a small, arbitrary number of applications on the same machine (currently a maximum of twenty-four client applications).

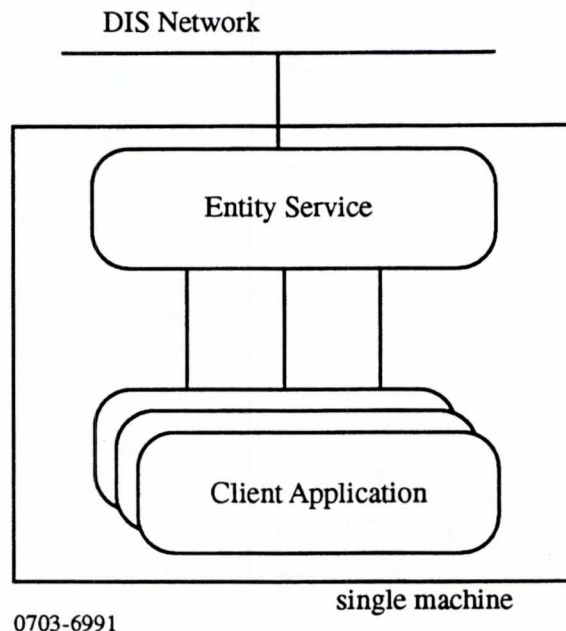


Figure 14. The logical relationship between the Entity Service and the client applications on a single machine.

The ripple effect is that the applications can be further decoupled from each other and more tightly encapsulated.

As a side note, we have bundled within the Entity Service the functionality to deal with certain miscellaneous DIS PDUs (such as fire and detonation). This functionality is currently fairly small and is included here merely for convenience. Should it become necessary later, some or all of these PDUs might be handled elsewhere in the system.

The Entity Service provides a mechanism for applications to log on and off. Logging on or off can occur at any point before or during a running scenario. The details of the procedure are handled internally, so that the client application programmer need not be concerned with them.

3.3.2.2 Segregation of Responsibility Between the Entity Service and the Client Applications

Responsibilities are divided between the Entity Service and the client applications in a manner that reflects both the goals of decoupling the applications from dealing with the network and centralizing dead reckoning within a given machine. The duties of the Entity Service include:

- Communication with the DIS network (this includes being able to read and write appropriate PDUs as well as adhering to DIS conventions associated with PDUs such as the five second rule for transmitting entity state)

- Providing meaningful yet generic forms for the transfer of entity related information between the service and the client applications
- Providing useful and flexible mechanisms for the client application to get to the information contained within the service
- Performing dead reckoning for all relevant entities on the DIS network (it is safe for the client applications to assume that the "most current" state information is available from the service)

Responsibilities of the client include:

- Properly making use of the mechanisms provided by the service
- Translating into and out of the generic form provided by the service and any specialized form needed by the client application (this is actually performed in the client program, but is below the abstraction level of the application programmer-- when the application sees data, it has been converted into whatever form is required)

The mindset of the application should be to assume that the Entity Service is "...a process that I can link up with at runtime to get the state information that I need."

3.3.2.3 Addressing the High Risk Areas

Coordinate conversion is an area that is not extremely difficult to deal with, but it is an area that draws information from and impacts other parts of the system. For example, the conversion of coordinates from geocentric to geodetic is fairly straightforward, but implies knowledge that DIS expects geocentric (which it does by definition) as well as knowledge that the application expects geodetic (which is not always the case). Perhaps the application expects a flat earth representation such as Universal Transverse Mercator (UTM). This is often accomplished by a conversion from geocentric to geodetic and then a conversion from geodetic to UTM.

Upon careful observation, many questions arise. Is a spherical earth model or an elliptical model required? If so, which ellipse? What about the conversion of vector quantities such as velocity? Is dead reckoning either computationally simpler or more accurate in any particular coordinate set? What about the case where one application expects geodetic coordinates and one expects UTM? Should the application's expectations even be known to the Entity Service? Most of these questions reinforce the claim that it should be the responsibility of the application to perform whatever conversions are necessary between the standard and what is needed internally.

In our implementation, a general coordinate conversion mechanism resides within the "EntityServiceClientInterface." Aside from initializing the conversion, its operation is transparent to the client application programmer. Details about the coordinate conversion mechanism used are found in the Coordinate Conversion Appendix document. It should be noted that with out implementation, future enhancements to the CC method can easily be swapped in to replace the existing method.

Identifying a data abstraction for entity information is an area that deserves further scrutiny. For now, the structure of the information passed between the service and the applications is similar, but not identical, to the DIS Entity State PDU.

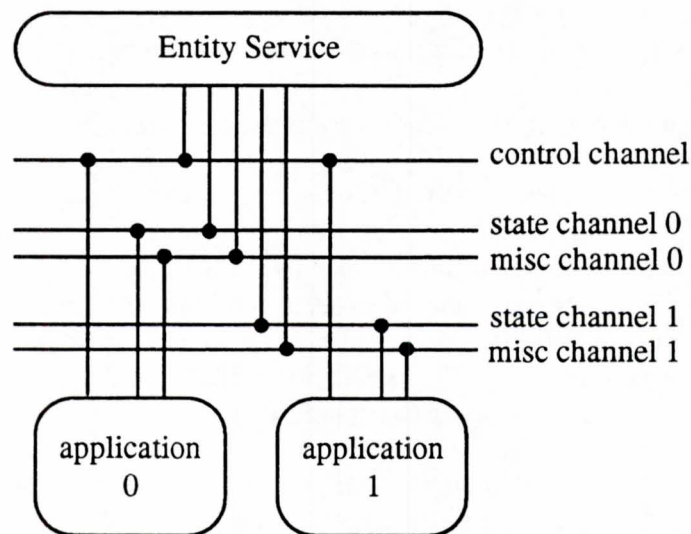
Interprocess communication is another area of concern. Conceptually, the links between the service and the client applications can be thought of as communications “channels”. These channels represent logical connections. See Figure below.

There is one “control” channel set up by the Entity Service upon its creation that is intended as the means of communicating control, or management information (information that is outside the scope of the running scenario). This version’s control centers around the applications logging on and off (i.e., establishing and breaking contact with the service).

There are also “state” and “miscellaneous” channels created for each application upon logging on to the Service. The state channel carries entity state information, while the miscellaneous channel carries things like fire and detonation messages.

Currently, the physical communication between the Entity Service and the client applications is carried out via shared memory protocols with semaphores and boolean flags for controlling access to the shared memories. However, this is actually an implementation detail that is transparent to the client application programmer. This physical implementation is subject to change in the future.

It is important to note that this version of the Entity Service is based on version 2.0.3 of the draft DIS standard. It is also important to note that this version of the Entity Service does *not* handle all of the PDUs specified in the standard, and as such should *not* be misinterpreted as “fully DIS-



0703-6990

Figure 15. The logical communication channels between the Entity Service and the client applications.

compliant". Specifically, this version of the Entity Service only handles only Entity State, Fire, and Detonation (standard) PDUs.

3.3.2.4 The Entity Service - A Client Programmer's View

Several software classes comprise the functionality provided by the Entity Service. These include classes for handling the network, the remote entity approximations, and the shared memory interface. It is important to note that one of these classes, the `EntityServiceClientInterface`, does not appear within the Entity Service. Instead, it is instantiated within the client application and is the client's end of the shared memory "wire." Basically, the ESCI insulates the application programmer from dealing with the shared memory and semaphore concerns. It is almost as if the application makes a function call (to the Entity Service) to get the information it needs.

3.3.2.4.1 Normal Operation of a Client Application

The normal operation of the client application interacting with the Entity Service during a simulation can be thought of as a client session. That is, the application "logs on" to the Entity Service, receives state and/or miscellaneous information as needed, and then "logs off" when complete.

The process of logging on actually happens in the constructor of the client interface. The client programmer simply needs to instantiate the `EntityServiceClientInterface`. Similarly, logging off is done in the destructor of the client interface. An obvious physical requirement is that the Entity Service process must be running prior to any client applications.

Logging on is handled via a "well known" shared memory. When the service begins operation, it allocates its control shared memory using a known shared memory key. Subsequently, when an application starts up, it attaches to the shared memory using the same known key and logs on to the service. This well known key is a configuration item. Logon structures within the control shared memory allow for application control information to flow between the service and the application. These same structures provide the detailed means for applications to log off when they are done.

Getting state information from the Entity Service is designed to be easy for the client application programmer. Basically, it simply requires an assertion by the client to the Entity Service to request a dump of state information. This request is followed by a period of time for the service to process the request, then a loop in which the client reads the available entity state information. This process is diagrammed and a code fragment is provided in the figures below.

Note that the client's assertion and wait are separated into two calls so parallelism is achieved. If the assert is followed immediately by the wait, then the client program will block until the service has processed the request. If, in the meanwhile, there is useful work for the client application to do, then the assert and the wait can straddle that work. If the Entity Service has completed the processing of the request prior to the wait call, then the wait returns immediately.

Recall that coordinate conversions into and out of any application-specific frame to the standard DIS geocentric are to be handled by the client application. However, these conversions are not

somewhere earlier...

```
client = new EntityServiceClientInterface("test", Hz, wellKnownKey,
                                         coordinateFrame, channelMask, errorCode);
```

somewhere in the main loop...

```
client->assertAwaitingState();           // alert service
    // other processing can be done in here...
client->waitForState();                  // data ready?
while (client->readState(&em)) {         // read one entity
    // do whatever...
}
```

Figure 16. A code fragment showing the state transfer interaction between the client application and the Entity Service.

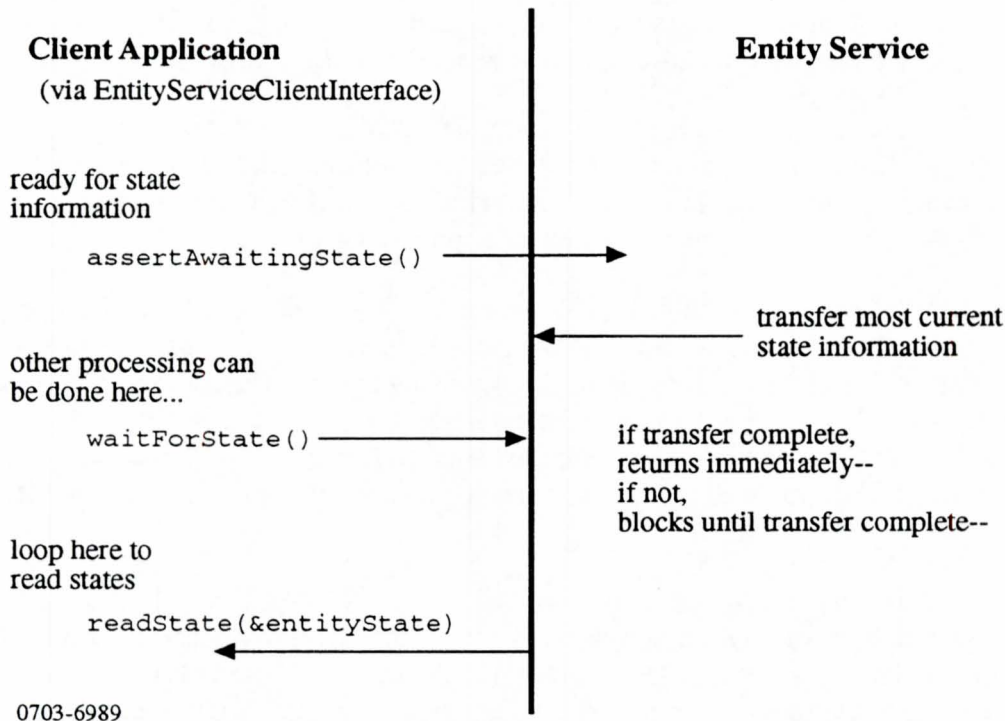


Figure 17. A schematic representation of the state transfer interaction between the client application and the Entity Service.

necessarily handled by the client programmer. A subset of conversions are provided by the EntityServiceClientInterface class. The client programmer needs only specify the form of the received data. The current version of the EntityServiceClientInterface supports three primary conversions--into and out of Flat Earth coordinates, Evans & Sutherland Image Generator (ESIG) coordinates, and Performer coordinates. Note that if a desired conversion is not directly supported, the client programmer can still receive data in DIS geocentric and perform the conversions himself.

Reading miscellaneous information from the Entity Service is handled in a similar fashion. An assertion is made for miscellaneous information, followed by a wait, followed by information read loops. See Figure below. For this version of the service, a bit mask is used in the constructor of the EntityServiceClientInterface class that specifies the types of miscellaneous information required by the client. The client simply calls the appropriate reading methods for the information.

```

somewhere earlier...
    client = new EntityServiceClientInterface("test", Hz, wellKnownKey,
        coordinateFrame, channelsMask, errorCode);

somewhere in the main loop...
    client->assertAwaitingState();           // alert service
        // other processing can be done in here...
    client->waitForMisc();                   // data ready?
    while (client->readFire(&fm)) {           // read one
        // do whatever...
    }
    while (client->readDetonation(&dm)) {     // read one
        // do whatever...
    }

```

Figure 18. A code fragment showing the miscellaneous transfer interaction between the client application and the Entity Service.

For situations in which a particular client application represents an active entity in a DIS simulation, then there are companion “writes” that are the reverse of the “reads” described above. The basic requirement is that when the application updates its local representation of the entity, it also writes the information to the Entity Service. It is also necessary for the client to inform the Service whenever it generates a fire or detonation. The Service takes care of informing others on the network.

Note that the client application is not concerned with when or how PDUs are broadcast across the network-- this is the responsibility of the Service. It is only required that the application inform the Service of changes in state, or any significant events such as fire and detonation.

3.3.2.4.2 The EntityServiceClientInterface Class

The EntityServiceClientInterface class represents the client application’s end of the shared memory connection with the Entity Service. The public interface to this class is described below.

```

class EntityServiceClientInterface
{
    public:
        // The constructor for the client interface is called with
        // parameters for the name of the application, the desired
        // frame rate of the application, the “well-known” shared

```

```

// memory key for logging on to the Entity Service, a
// descriptor of the coordinate conversion method needed,
// a bit-mask describing which messages the application
// desires (currently, values of ES_WANT_ENTITY_STATES,
// ES_WANT_FIRES, and ES_WANT_DETONATIONS can be ORed
// together to specify which messages the client wants
// sent to it), and a parameter used for notifying the
// application about errors during the logon process. The
// client interface will, during the constructor, attempt
// to log on to the Entity Service. Failure to log on to
// the service will result in a fatal error.
// (Make sure the Service is running first)
EntityServiceClientInterface( char *appName,
                              int appFrameRate,
                              key_t controlShmkey,
                              int coordinateFrame,
                              int channelsMask,
                              int *errorCode );

// The destructor is responsible for logging off of the
// Entity Service and cleaning up.
~EntityServiceClientInterface();

// The next four routines provide reading and writing
// capability to the client application for state
// information. Note that the reading functions should
// only be used if ES_WANT_ENTITY_STATES was specified in
// the channelsMask of the constructor.
// -----
// The assertAwaitingState call is the signal to the
// Entity Service that the client application wants
// the "most current state" of all the entities.
int    assertAwaitingState();

// The waitForState call delays the read of state
// information until the service has finished processing
// the request.
int    waitForState();

// The readState call is looped on by the client
// application to get the "most current state" for each
// entity. The readState call returns NULL when all
// entities have been read.
int    readState( EntityMsg *msg );

// The writeState call is used by a client application that
// represents an active entity in the scenario. When the
// client updates the state of the entity, the client must
// write that new state to the service.
// Note that this call is standalone, no assertion is
// necessary.
int    writeState( EntityMsg *msg );

```



```

// The next six routines provide reading and writing
// capability to the client application for miscellaneous
// information. Note that the reading functions should
// only be used if the appropriate bit mask (ES_WANT_FIRES
// or ES_WANT_DETONATIONS) was specified in the
// channelsMask of the constructor.
// -----
// The assertAwaitingMisc call is the signal to the Entity
// Service that the client application needs any
// miscellaneous information.
int    assertAwaitingMisc();

// The waitForMisc call delays the read of miscellaneous
// information until the service has finished processing
// the request.
int    waitForMisc();

// The readFire call is looped on by the client
// application to get any new fire messages. The readFire
// call returns NULL when all fire messages have been
// read.
int    readFire( FireMsg *msg );

// The readDetonation call is looped on by the client
// application to get any new detonation messages. The
// readDetonation call returns NULL when all detonation
// messages have been read.
int    readDetonation( Detonation *msg );

// The writeFire call is used by a client application when
// an entity it represents fires a weapon.
// Note that this call is standalone, no assertion is
// necessary.
int    writeFire( FireMsg *msg );

// The writeDetonation call is used by a client application
// when an entity it represents detonates a munition.
// Note that this call is standalone, no assertion is
// necessary.
int    writeDetonation( DetonationMsg *msg );

```

3.3.2.4.3 The Structures Used to Communicate with the EntityService-ClientInterface

The structures used in both read and write calls to the EntityServiceClientInterface are detailed below.

```

// State Message
typedef struct
{
    ESPDUHeader    header;
    EntityID       id;

```

```

    u_char        forceID;
    u_char        numArticulatedParts;
    EntityType    type;
    EntityType    altType;
    Vector        linearVelocity;
    double        x, y, z;
    float         h, p, r;
    u_long        appearance;
    DRParameters  dr;
    u_char        characterSet;
    char          desc[12];
    u_long        capabilities;
    ArtMsg        articulatedParts[MAX_ARTICULATED_PARTS];
} EntityMsg;

// Fire Message
typedef struct
{
    ESPDUHeader    header;
    EntityID       attackerID;
    EntityID       targetID;
    EntityID       munitionID;
    EventID        eventID;
    double         x, y, z;
    EntityType     burst;
    u_short        warhead;
    u_short        fuze;
    u_short        quantity;
    u_short        rate;
    Vector         velocity;
    float          range;
} FireMsg;

// Detonation Message
typedef struct
{
    ESPDUHeader    header;
    EntityID       attackerID;
    EntityID       targetID;
    EntityID       munitionID;
    EventID        eventID;
    Vector         velocity;
    double         worldX, worldY, worldZ;
    EntityType     burst;
    u_short        warhead;
    u_short        fuze;
    u_short        quantity;
    u_short        rate;
    float          entityX, entityY, entityZ;
    u_char         result;
    u_char         numArticulatedParts;
    ArtMsg        articulatedParts[MAX_DETONATION_PARTS];
} DetonationMsg;

```

3.3.2.4.4 The Configuration File

The Entity Service requires a configuration file in order to operate. The file has six variables that can be set:

- `eservHz n`
where n is the number of times per second that the Service should process entities and miscellaneous information.
- `port n`
where n is the port number to use for broadcast network communication.
- `maxClients n`
where n is the maximum number of clients that this service should provide for.
- `smoothing t`
where t is either *snapping*, *linear* or *hermite* to indicate which smoothing method to use. See the Dead Reckoning appendix.
- `controlShmkey n`
where n is the “well-known” shared memory key for the control channel.
- `nextShmkey n`
where n is the first shared memory key to use for client channels (state and miscellaneous)
- `deltaShmkey n`
where n is the value to add to nextShmkey for each channel after each client logs on (i.e. the difference between successive channel shared memory keys).

A simple configuration file might look like:

```
ESERV_CONFIG_FILE

eservHz    40.0

port       3000

maxClients 10

smoothing  hermite

controlShmkey  0x00004000
nextShmkey     0x00008000
deltaShmkey    0x00000010
```

Note that the first line (`ESERV_CONFIG_FILE`) is required. Once a configuration file is created, run the Entity Service by placing the name of the configuration file as a command-line parameter to the Entity Service. For example, “entityServ eserv.cfg” will run the Entity Service using a configuration file named `eserv.cfg`.

3.3.2.4.5 The Configuration File for Coordinate Conversion

As mentioned earlier, coordinate conversions into and out of the DIS standard are conducted within the client application. However, the idea is for the client interface to perform any necessary

conversion prior to the client application receiving the data. This meets the strategic goal of placing conversion responsibility within the client application, and at the same time insulates the client programmer from explicitly performing the conversions.

In order to facilitate the conversions, there must be a configuration file present that contains the necessary information. By convention, the file name is

CC.<name>.cfg where <name> is flat_earth, esig, or performer

The file format is in a free form that currently supports three basic types of information that need to be in the file. The necessary pieces of information are:

1. geocentricOffset <x> <y> <z>
2. toDIS

 <M[0,0]> <M[0,1]> <M[0,2]>
 <M[1,0]> <M[1,1]> <M[1,2]>
 <M[2,0]> <M[2,1]> <M[2,2]>
3. toFlatEarth

 <M[0,0]> <M[0,1]> <M[0,2]>
 <M[1,0]> <M[1,1]> <M[1,2]>
 <M[2,0]> <M[2,1]> <M[2,2]>

The geocentric offset is used to translate global coordinates into a flat earth coordinate system, and the toDIS and toFlatEarth are used to rotate the coordinate frames so they match with either the DIS coordinate view or the Flat Earth view. Currently these matrices are not automatically generated, and therefore need to be created by external means.

3.3.2.5 Lessons Learned in this Iteration

Several lessons were learned in this second iteration of the design of the Entity Service. The knowledge gained serves as input to the next iteration of the design. These lessons are mentioned below.

3.3.2.5.1 Coordinate Conversions

The phrase coordinate conversion contains something of a paradox. At the surface, things like changes of base are fairly straightforward, but there is much subtlety (and room for error) below the surface. Assumptions are inherent in any coordinate conversion. These assumptions are often masked, and sometimes even lost over a series of transformations. It sometimes also happens that certain of the assumptions conflict over such a series. Differences in coordinate conversion methods serve as one of the causes of correlation problems. This version of the Entity Service contains

only the conversions necessary for DT research, but provides a common mechanism for implementing any conversion that might be needed in the future.

3.3.2.5.2 The Data Abstraction

As mentioned above, identifying a data abstraction for entity information is an area that deserves further scrutiny. The cause for revisiting entity information is primarily the increasing fidelity of the physical models associated with the platform entities. Adding capabilities like trafficability and mobility calculations to a vehicle model not only requires that more inputs about the terrain be available, but also that more inputs about the physical characteristics of that vehicle be available. It is not clear, but it is at least plausible, that some of these additional inputs might be required for transmission to newer dead reckoning models.

The Entity Service begins to open the possibility for experimenting with higher fidelity vehicle models and dead reckoning models. Evidence will be required to justify any call for adding fields to the entity state PDU or for adding new dead reckoning models. We have created a fairly robust mechanism for such experimentation by encapsulating the details of the exchange between the Entity Service and the client application.

3.3.2.5.3 Interprocess Communication

Interprocess communication is an area that has major ramifications with respect to both complexity and performance. In the Entity Service, the shared memory/semaphore/boolean flag approach was chosen as the compromise that provided for sufficient performance without complexity in the software. The most difficult problem was to effectively encapsulate access to the shared memory to make the job of the application programmer easy. The current version is not as clean as desired, but it insulates the application programmer from the details of the interprocess communication while providing flexibility.

3.3.3 Software Design

Figure 19 shows the Booch diagram of classes used in the Entity Service. In order to better facilitate the iterative design process as well as maintenance, the classes involved were abstracted into a hierarchy so that common implementation details can exist in one location. In fact, all three services (Entity, Terrain and Fluid) are based on the same core classes.

3.3.4 Programmer's Guide

A guide to the programmer of the Entity Service is intuitively divided into two sections: one for a "client application programmer" that contains the detail of the client interface, and one for a "service programmer" that contains details on semaphores, shared memory, network interface, "channels," structure of the service, etc.

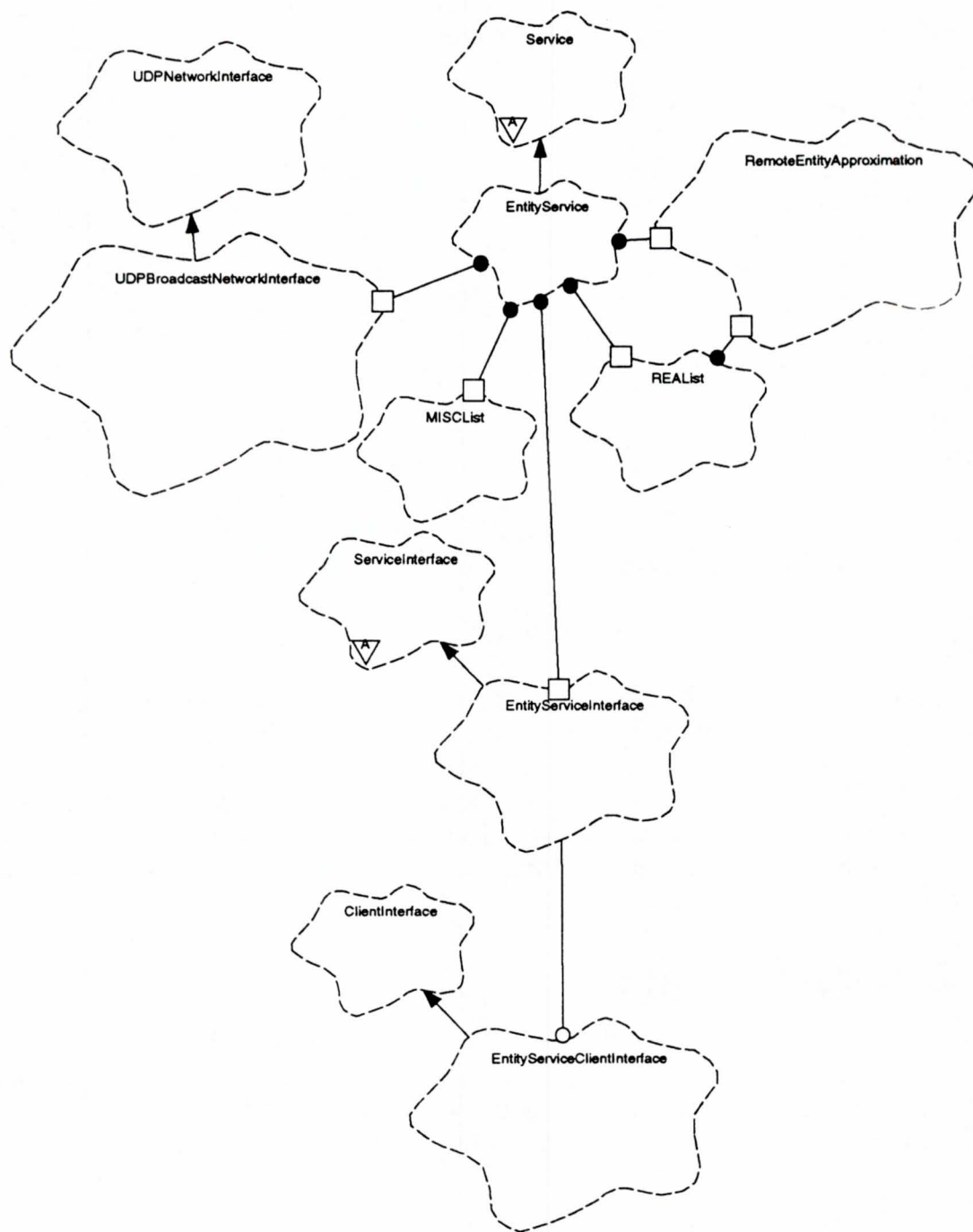


Figure 19. Booch Diagram of classes in the Entity Service

3.3.4.1 Service Programmers

In order to spend more time discussing the structure of the Service and the “channels,” basic knowledge of semaphores, shared memory and network interfacing is assumed. Since the service and the client applications exist as separate processes, they communicate through common struc-

tures in shared memory. Semaphores are used to control access to these common data structures including the control structure (for logging on and off) as well as the state and miscellaneous channels for each client application. The network interface is used to communicate with other machines (reading and writing of PDUs).

In order to provide the capability of communication between the Service and the client applications, a number of logical "channels" were developed as mentioned earlier. Each channel provides a pathway for one type of data. For example, the Entity Service uses three types of channels: one for control information, one for entity states and one for all miscellaneous (fire and detonation) information. The structure of each channels appears as follows:

```
typedef struct
{
    int      inUse;
    pid_t    appPid;
    char      appName[80];
    int      appFrameRate;
    int      wantsToLogon;
    int      wantsToLogoff;
    int      appIndex;
    int      channelsMask;
    int      variableSize;
    int      loggedOn;
    key_t     accessShmkey;
    key_t     channelShmkeys[MAX_CHANNELS];
    key_t     variableBlockShmkeys[MAX_CHANNELS];
} LogonStruct;

// Control Channel
typedef struct
{
    int      numberToLogon;
    int      numberToLogoff;
    int      maxApplications;
    LogonStruct  logon[MAX_APPLICATIONS];
} ServiceControlChannel;

// State Channel
typedef struct
{
    int      numInbound;
    int      nextInboundRead;
    int      nextInboundWrite;
    DISEntityStatePDU  inbound[MAX_INBOUND_ENTITIES];
    int      numOutbound;
    int      nextOutbound;
    DISEntityStatePDU  outbound[MAX_OUTBOUND_ENTITIES];
} EntityServiceStateChannel;

typedef struct
{
```

```

        int                numInbound;
        int                nextInboundRead;
        int                nextInboundWrite;
        DISFirePDU         inbound[MAX_INBOUND_FIRE];
        int                numOutbound;
        int                nextOutbound;
        DISFirePDU         outbound[MAX_OUTBOUND_FIRE];
    } FireChannel;

typedef struct
{
    int                numInbound;
    int                nextInboundRead;
    int                nextInboundWrite;
    DISDetonationPDU   inbound[MAX_INBOUND_DETONATION];
    int                numOutbound;
    int                nextOutbound;
    DISDetonationPDU   outbound[MAX_OUTBOUND_DETONATION];
} DetonationChannel;

// Miscellaneous Channel
typedef struct
{
    FireChannel         fire;
    DetonationChannel   detonation;
} EntityServiceMiscChannel;

```

Each channel basically consists of two lists of information (inbound and outbound). Note that the two types of miscellaneous information have been combined into one channel (EntityServiceMiscChannel). This was done in order to combine these logically since they are more event-based information than state-based. If warranted in the future, this channel could be moved into a separate "Event Service" as two separate channels.

3.3.4.2 Client Programmers

As mentioned earlier, the client interface provides a means of information hiding. The client applications should not have to deal with issues such as shared memory communication and semaphores. This is all hidden in the client interface. Furthermore, it is important to note that the client interface also contains any information specific to the individual client. For example, in the Entity Service, coordinate conversions were placed within the EntityServiceClientInterface since the specific coordinate system used is related to the client instantiating the class. By the time the Service gets the data, it has already been transformed into geocentric (as required by DIS).

4.0 Terrain Service

4.1 Introduction to Terrain Service

Similar to how the Entity Service interfaces between the client applications and the state of entities in the shared environment, a Terrain Service is needed to serve as the interface between the client applications and the state of the terrain in the shared environment. For more details on the Entity Service, see the EntityService documentation.

4.1.1 Background

In distributed interactive simulations, it is necessary for each entity to have a logical view of the terrain, and if the terrain is dynamic, all of the representations for the terrain are the same within a specified tolerance. Thought of in these terms, the several attributes of the terrain are yet another component of the shared environment.

As discussed briefly in the Entity Service document, many of the functionalities associated with the dynamic terrain effort are individually relatively small. In particular, cratering, soil slumping, and others are far too small to warrant allocation of a machine. This dilemma leads to two fundamental approaches. Many, or perhaps all, of these small dynamic terrain functionalities could be bundled into one large eclectic application, or each of these small functionalities could reside in separate applications.

Because development and maintenance would be unnecessarily complicated by the increased coupling and weaker encapsulation of the dynamic terrain functionalities if they were bundled into a single application, we chose to split the functionalities into separate applications. As discussed further below and in other parts of the documentation, we have found significant expressive power based on this decision.

It is important that a shared environment exists, and it would be ideal if applications were insulated from the duties of managing that shared environment. This has been thematic in our approach. Better encapsulation of the environment, and the applications, not only simplifies development, but also helps to localize the impact of adding new capabilities in the future. Whether a dynamic terrain application or a more "standard" simulator, the software can be dramatically easier to build and maintain with prudent use of abstraction and encapsulation.

4.1.2 Problem Statement

The requirement for the Terrain Service is a single application that can perform Dynamic Terrain I/O (input/output), maintain state for an arbitrary number of terrain attributes, and support several simultaneous client applications that require independent subsets of this terrain state information.

4.1.3 Solution

Several design goals are related to the Terrain Service, including:

- Decouple the application from the form of the data - This is imperative with respect to the design of a flexible architecture. It is not safe to assume that the data forms currently in use will necessarily be used in the future, nor is it safe to assume that new desired functionalities will necessarily fit any currently used form. In this case, the form referred to is the communications protocol (prototypical DT PDUs). With our implementation the client applications do not know, nor should they know, what the underlying terrain database looks like. The mindset is different. *No longer do applications think in terms of what they can get out of the database, but rather they think in terms of what they want from the database.*
- Find effective data abstractions for relaying terrain information - Decoupling the application from the form of the data requires the development of effective data abstractions that can be used as intermediaries.
- Support a small, but arbitrary number of applications - To provide for future demands, no artificial constraints should be placed on the number of client applications that can be running on a single machine at any given time. The principal constraints should be performance requirements and resource availability. If twenty lightweight applications could perform on a single machine, it should be possible to allocate all twenty of those to a single machine.
- Be Flexible - The Terrain Service should be flexible enough to handle changes to the protocols (both between the service and the network, and between the service and the applications). Future requirements may require additional state information for the terrain. A sandbox for testing experimental protocol changes is required.

4.2 Analysis

The cornerstone of our approach is the use of object oriented analysis and design techniques. The object oriented paradigm is characterized by an iterative, incremental approach in which both the understanding of the problem and the design of a solution for that problem evolve over time from the general towards the specific. It is important to get results early, and to use the knowledge gained to refine subsequent iterations.

As part of our iterative approach, we have chosen to enumerate and address high-risk areas as early as possible. In the early iterations, we want to face the issues that have the highest potential to cause radical changes in the design. In some sense, we want to build a reasonable skeleton before we try to flesh it out. Areas identified early as high-risk for the Terrain Service include:

- Active terrain database - What sort of database representation will allow for dynamic updates at runtime? Can the database make use of standard data sources and still have enough expressive power to handle arbitrary resolution? Can the database be flexible enough to add unforeseen attributes as required?
- Identifying a data abstraction for terrain state information - What is the form of the data exchanged between the service and the client applications? This abstraction ultimately becomes the sandbox for experimental protocols. What are possible/reasonable/useful ways to handle this experimentation?
- Interprocess communications - What are possible/reasonable/useful ways to handle communication between the service and the client applications? What are the tradeoffs between performance and complexity?

4.3 Implementation

4.3.1 Problem Statement

Taking into account the stated design goals and the desire to address the high-risk areas early, a single application is needed to serve as the interface between the client applications and the shared environment for terrain state information.

4.3.2 User's Guide

4.3.2.1 The Structure of the Relationship

The Terrain Service as we've defined it runs in its own process and serves as the intermediary between the client applications and the network. There is one copy of the Terrain Service per physical machine. See Figure 20.

We have encapsulated the functionalities of communication and maintenance of terrain state within the Terrain Service. We have also gained the capability to support a small, arbitrary number of applications on the same machine (currently a maximum of twenty-four client applications). The ripple effect is that the applications can be further decoupled from each other and more tightly encapsulated.

The Terrain Service provides a mechanism for applications to log on and off. Logging on or off can occur at any point before or during a running scenario. The details of the procedure are handled internally, so that the client application programmer does not need to be concerned with them.

4.3.2.2 Segregation of Responsibility Between the Terrain

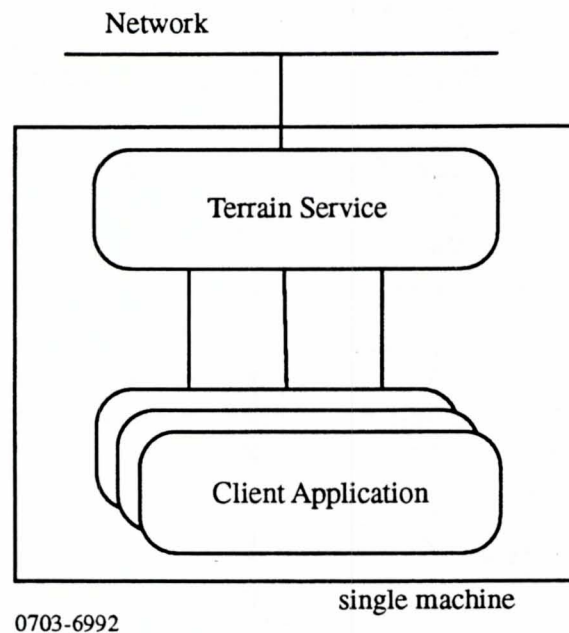


Figure 20. The logical relationship between the Terrain Service and the client applications on a single machine.

Service and the Client Applications

Responsibilities are divided between the Terrain Service and the client applications in a manner that reflects the goal of decoupling the applications from having to deal with the network. The duties of the Terrain Service include:

- Communication with other services via the network (this includes being able to read and write prototypical PDUs)
- Providing meaningful yet generic forms for the transfer of terrain related information between the service and the applications
- Providing useful and flexible mechanisms for the client application to get to the information contained within the service
- It is safe for the client applications to assume that the “most current” terrain state information is available from the service

Responsibilities of the client include:

- Properly making use of the mechanisms provided by the service
- Translating into and out of the generic form provided by the service and any specialized form needed by the client application (if required, this coordinate conversion would be handled with the same means as is found in the Entity Service)

The mindset of the application should be to assume that the Terrain Service is "...a process that I can link up with at runtime to get the terrain state information that I need."

4.3.2.3 Addressing the High Risk Areas

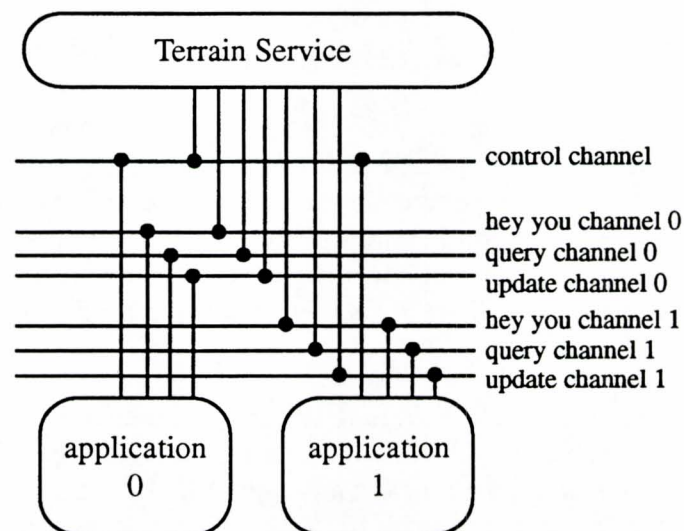
Developing a database that can represent the terrain in a flexible way is important. Furthermore, the ability to store multiple terrain attributes is a key aspect when considering vehicle mobility. Refer to the Dynamic Terrain Database document for more information.

Identifying a data abstraction for terrain information is an area that deserves further scrutiny. For now, the structure of the information passed between the service and the applications is similar but not identical to the prototypical Terrain State PDU.

Interprocess communication is another area of concern. Conceptually, the links between the service and the client applications can be thought of as communications "channels". These channels represent logical connections. See the figure below.

There is one "control" channel set up by the Terrain Service upon its creation that is intended as the means of communicating control, or management information (information that is outside the scope of the running scenario). This version's only control centers around the applications logging on and off (i.e., establishing and breaking contact with the service).

There are also "hey you," "query" and "update" channels created for each application when it logs on to the service. The hey you channel is the one that carries terrain update information that has been received from the network or other client applications, the query channel is used to sup-



0703-6988

Figure 21. The logical communication channels between the Terrain Service and the client applications.

port terrain attribute queries from clients, and the update channel carries a terrain update from the client application.

Currently, the physical communication between the Terrain Service and the client applications is carried out via shared memory protocols with semaphores and boolean flags for controlling access to the shared memories. However, this is actually an implementation detail that should not be considered by the client application programmer. The physical implementation is subject to change in the future.

4.3.2.4 The Terrain Service - A Client Programmer's View

Several software classes comprise the functionality provided by the Terrain Service. These include classes for handling the network, the active database, and the shared memory interface. It is important to note that one of these classes, the `TerrainServiceClientInterface`, does not appear within the Terrain Service. Instead, it is instantiated within the client application and serves as the client's end of the shared memory "wire." Basically, it insulates the application programmer from dealing with the shared memory and semaphore/boolean flag concerns. It is almost as if the application makes a function call (to the Terrain Service) to get the information it needs.

4.3.2.4.1 Normal Operation of a Client Application

The normal operation of the client application interacting with the Terrain Service can be thought of as a client session. That is, the application "logs on" to the Terrain Service, receives terrain state information as needed, and then "logs off" when complete.

The process of logging on actually happens in the constructor of the client interface. All the client programmer needs to do here is to instantiate the `TerrainServiceClientInterface`. Similarly, logging off is done in the destructor of the client interface. An obvious physical requirement is that the Terrain Service process must be running prior to any client applications.

Logging on is handled via a "well known" shared memory. That is, when the service starts up it allocates its control shared memory using a known shared memory key. Subsequently, when an application starts up, it attaches to the shared memory using the same known key and logs on to the service. This well known key is a configuration item. Logon structures within the control shared memory allow for application control information to flow between the service and the application. These same structures provide the detailed means for applications to log off when they are done.

Getting terrain state information from the Terrain Service is designed to be as painless to the client application programmer as possible. All that is required is an assertion by the client to the Terrain Service to query for state information, and a period of time for the service to process the request. Then the client reads the terrain state information. This process is diagrammed in Figure 23, and a code fragment is provided in Figure 22.

Note that the assertion and the wait are separated into two calls from the client so that some parallelism can be achieved. If the assert is followed immediately by the wait, then the client program

somewhere earlier...

```
client = new TerrainServiceClientInterface("test", Hz, wellKnownKey,
channelsMask, maxAttributes, maxRows, maxColumns,
errorCode);
```

somewhere in the main loop...

```
client->assertAwaitingHeyYou();           // alert service
// other processing can be done in here...
client->waitForHeyYou();                   // data ready?
while (client->readHeyYou(&em)) {          // read one entity
    // do whatever...
}
```

Figure 22. A code fragment showing the state transfer interaction between the client application and the Terrain Service.

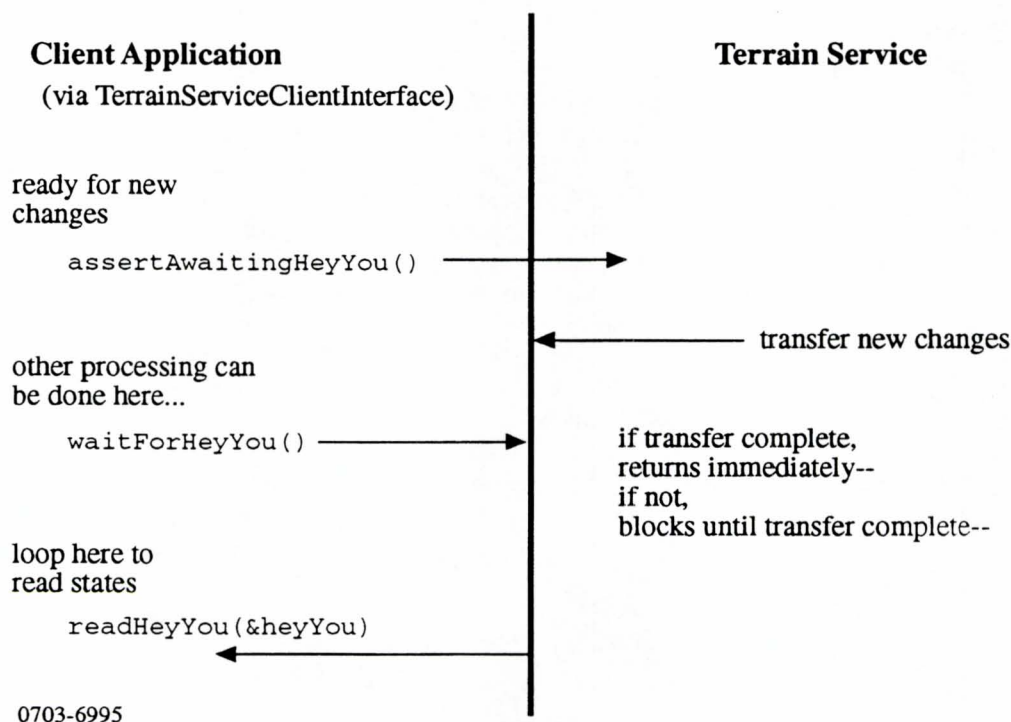


Figure 23. A schematic representation of the “terrain changed” transfer interaction between the client application and the Terrain Service.

will block until the service has processed the query. If, on the other hand, there is useful work that the client application can be doing in the meantime, then the assert and the wait can straddle that other work. If the Terrain Service has completed the processing of the query prior to the wait call, then the wait returns immediately.

For situations where a particular client application wants to make a change to a terrain attribute, there is a companion “write” that is the reverse of the “read” described above. The basic require-

ment is that when the application wants to update the terrain, it writes that information to the Terrain Service.

Note that the client application does not need to be concerned with when or how PDUs must be broadcast to the network -- that is the responsibility of the service. It is only required that the application inform the service of desired changes in state.

4.3.2.4.2 The TerrainServiceClientInterface Class

The TerrainServiceClientInterface class represents the client application's end of the shared memory connection with the Terrain Service. The public interface to this class is described below.

```
class TerrainServiceClientInterface
{
    public:
        // The constructor for the client interface is called
        // with parameters for the name of the application, the
        // desired frame rate of the application, the
        // "well-known" shared memory key for logging on to the
        // Terrain Service, a bit-mask describing which messages
        // the application will ask for (currently, only the value
        // TS_WANT_HEY_YOUS is available), a maximum number of
        // attributes that will be queried or updated at one time,
        // a maximum number of rows that will be queried or
        // updated at one time, a maximum number of columns that
        // will be queried or updated at one time and a parameter
        // to use for notifying the application about errors
        // during the logon process. The client interface will,
        // during the constructor, attempt to log on to the
        // Terrain Service. Failure to log on to the service will
        // result in a fatal error.
        // (Make sure the service is running first)
        TerrainServiceClientInterface( char *appName,
                                       int appFrameRate,
                                       key_t controlShmkey,
                                       int channelsMask,
                                       int maxAttributes,
                                       int maxRows,
                                       int maxColumns,
                                       int *errorCode );

        // The destructor is responsible for logging off of the
        // Terrain Service and cleaning up.
        ~TerrainServiceClientInterface();

        void    allocateMessage( AttributeMsg *msg,
                                int blockIndex, int rows,
                                int columns );

        void    deallocateMessage( AttributeMsg *msg,
                                int blockIndex );
}
```

```

// The next three routines provide reading capability to
// the client application for hey you (a piece of terrain
// changed) information. Note that these should only be
// used if TS_WANT_HEY_YOUS was specified in the
// channelsMask in the constructor.
// -----
// The assertAwaitingHeyYou call is the signal to the
// Terrain Service that the client application needs to
// get the new notifications of changed terrain.
int      assertAwaitingHeyYou();

// The waitForHeyYou call delays the read of hey you
// information until the service has finished processing
// the request.
int      waitForHeyYou();

// The readHeyYou call is looped on by the client
// application to get the new notifications of changed
// terrain. The readHeyYou call returns NULL when all
// notifications have been read.
int      readHeyYou( AttributeMsg *msg );

// The next three routines provide querying capability to
// the client application for the terrain database.
// -----
// The assertAwaitingQuery call is the signal to the
// Terrain Service that the client application needs to
// get (query) a piece of the terrain database. The
// parameter tells the Terrain Service which portion of
// the database to query and at what resolution.
int      assertAwaitingQuery( AttributeMsg *msg );

// The waitForQuery call delays the read of the returned
// query information until the service has finished
// processing the request.
int      waitForQuery();

// The readQuery call places the returned data from the
// query in the parameter.
int      readQuery( AttributeMsg *msg );

// The writeUpdate call is used by a client application
// to perform an update to the terrain database.
// Note that this call is standalone, no assertion is
// necessary.
int      writeUpdate( AttributeMsg *msg);
};

```

4.3.2.4.3 The Structures Used to Communicate with the TerrainSer-

viceClientInterface

The structures used in the read and write calls to the TerrainServiceClientInterface are detailed below.

```
typedef struct
{
    u_char    attribute;
    u_char    plane;
    float     p1[2];           // Clockwise starting at p1
    float     p2[2];
    float     p3[2];
    float     p4[2];
    u_char    numberOfRows;    // p1..p4 direction
    u_char    numberOfColumns; // p1..p2 direction
    float     *data;
} MsgData;

// Attribute Message
typedef struct
{
    TSPDUHeader    header;
    u_char          numberOfBlocks;
    MsgData         block[TS_MAX_BLOCKS];
} AttributeMsg;
```

4.3.2.4.4 The Configuration File

The Terrain Service requires a configuration file in order to operate. The file has eight variables that can be set:

- **tservHz n**
where n is the number of times per second that the Service should process entities and miscellaneous information.
- **dtddbConfigFilename n**
where n is the filename of the configuration file for the Dynamic Terrain Database (DTDB)
- **maxClients n**
where n is the maximum number of clients that this service should provide for.
- **controlShmkey n**
where n is the “well-known” shared memory key for the control channel.
- **nextShmkey n**
where n is the first shared memory key to use for client channels (hey you, query and update)
- **deltaShmkey n**
where n is the value to add to nextShmkey for each channel after each client logs on (i.e. the difference between successive channel shared memory keys).
- **netConfig $t k$**
where t is one of “server”, “client”, “relay” or “distributed” that defines in what mode the Service will operate, and k is the key of the link that goes toward a central server (note that k is only used in a Service running in “relay” mode). “Modi-

ying the TS Configuration” on page 242 discusses the different network configurations in detail.

- link $k\ i\ p$
where k is a unique key assigned to this link, i is the IP address of machine to connect with, and p is the port number to use for the connection.

A simple configuration file might look like:

```
TSERV_CONFIG_FILE

tservHz      40.0

dtddbConfigFilename  dtddb.cfg

maxClients    10

controlShmkey  0x00005000
nextShmkey     0x00009000
deltaShmkey    0x00000010

netConfig     distributed

link  0  127.0.0.1  4000
```

Note that the first line (TSERV_CONFIG_FILE) is required. Once you have a configuration file, you run the Terrain Service by placing the name of the configuration file as a command-line parameter to the Terrain Service. For example, “terrainServ tserv.cfg” will run the Terrain Service using a configuration file named tserv.cfg.

4.3.2.5 Lessons Learned in this Iteration

Several lessons were learned in this first iteration of the design of the Terrain Service. This knowledge gained functions as input to the next iteration of the design. The major lessons are mentioned below.

4.3.2.5.1 The Data Abstraction

As mentioned above, identifying a data abstraction for terrain information is an area that deserves further scrutiny. The cause for revisiting terrain information is primarily to increase fidelity of the physical models associated with the platform entities. Adding capabilities like trafficability and mobility calculations to a vehicle model requires that more inputs about the terrain be available.

4.3.2.5.2 Interprocess Communication

Interprocess communication is an area that can have major ramifications with respect to both complexity and performance. In this version of the Terrain Service, the shared memory/semaphore approach was chosen as the compromise that provided for good performance without extreme complexity in the software. The most difficult problem was to effectively encapsulate

access to the shared memory so as to make the job of the application programmer as easy as possible. The current version is not as clean as it needs to be, but it does pretty well insulate the application programmer from the details of the interprocess communication while providing a reasonable level of flexibility.

4.3.3 Software Design

Figure 24 shows the Booch diagram of classes used in the Terrain Service. In order to better facilitate the iterative design process as well as maintenance, the classes involved were abstracted into a hierarchy so that common implementation details can exist in one location. In fact, all three services (Entity, Terrain and Fluid) are all based on the same core classes.

4.3.4 Programmer's Guide

A guide to the programmer of the Terrain Service is intuitively divided into two sections: one for a "client application programmer" that contains the details of the client interface, and one for a "service programmer" that contains details on semaphores, shared memory, network interface, "channels," structure of the service, etc. However, it is important to note that the Terrain Service uses Unicast network connections because the size of the prototype PDU is too large for broadcast methods.

4.3.4.1 Service Programmers

In order to spend more time discussing the structure of the service and the "channels," basic knowledge of semaphores, shared memory and network interfacing is assumed. Since the service and the client applications exist as separate processes, they communicate through common structures in shared memory. Semaphores are used to control access to these common data structures including the control structure (for logging on and off) as well as the state and miscellaneous channels for each client application. The network interface is used to communicate with other machines (reading and writing of PDUs).

In order to provide the ability to communicate between the service and the client applications, a number of logical "channels" were developed (as mentioned earlier). Each channel provides a pathway for one type of data. For example, the Terrain Service uses four types of channels: one for control information, one for "hey yous" (terrain change notifications), one for queries and one for updates. The structure of each channel appears as follows:

```
typedef struct
{
    int      inUse;
    pid_t    appPid;
    char     appName[80];
    int      appFrameRate;
    int      wantsToLogon;
    int      wantsToLogoff;
    int      appIndex;
    int      channelsMask;
```

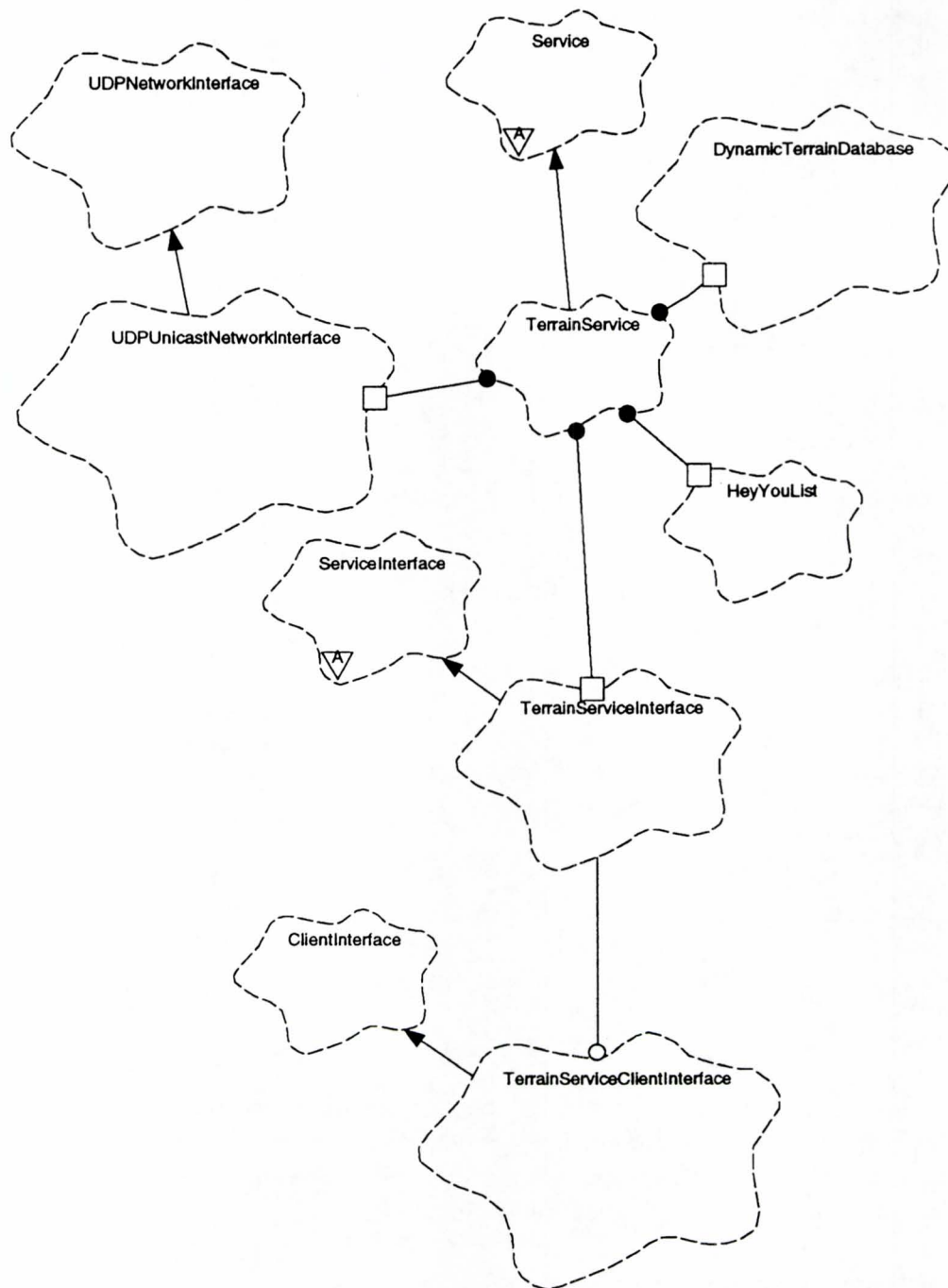


Figure 24. Booch diagram of classes for the Terrain Service


```

        int         variableSize;
        int         loggedOn;
        key_t       accessShmkey;
        key_t       channelShmkeys[MAX_CHANNELS];
        key_t       variableBlockShmkeys[MAX_CHANNELS];
    } LogonStruct;

// Control Channel
typedef struct
{
    int         numberToLogon;
    int         numberToLogoff;
    int         maxApplications;
    LogonStruct logon[MAX_APPLICATIONS];
} ServiceControlChannel;

// Hey You Channel
typedef struct
{
    AttributeGram inbound[TS_MAX_INBOUND_HEY_YOUS];
    int         numInbound;
    int         nextInboundRead;
    int         nextInboundWrite;
} HeyYouChannel;

// Query Channel
typedef struct
{
    AttributeGram inbound[TS_MAX_INBOUND_QUERIES];
    int         numInbound;
} QueryChannel;

// Update Channel
typedef struct
{
    AttributeGram outbound[TS_MAX_OUTBOUND_UPDATES];
    int         numOutbound;
} UpdateChannel;

```

Note that the HeyYou and Update channels only contain one list each since they are unidirectional channels (HeyYous only go towards the client applications since they are meant to notify them of terrain changes, and Updates only go towards the Terrain Service since they are meant to tell the service of an update that the client application wants to make).

4.3.4.2 Client Programmers

As mentioned earlier, the client interface provides a means of information hiding. The client applications should not have to deal with issues such as shared memory communication, semaphores, etc. This is all hidden away in the client interface. Furthermore, it is important to note that the client interface can also contain any information specific to the individual client. For example, coordinate conversions of the terrain updates could be done within the TerrainService-ClientInterface if needed or wanted in the future. Therefore, by the time the service would get the

data, it will have been already transformed into geocentric (as required by DIS).

5.0 Fluid Service

5.1 Introduction to Fluid Service

Similar to how the Entity Service interfaces between the client applications and the state of entities in the shared environment, a Fluid Service was developed to serve as the interface between the client applications and the state of the Fluid in the shared environment. For more details on the Entity Service, see the Entity Service documentation.

5.1.1 Background

In distributed interactive simulations, it is necessary for each entity to have a logical view of the fluid, and if the fluid is dynamic, all of the representations for the fluid are the same within a specified tolerance. Thought of in these terms, the several important attributes of the fluid are yet another component of the shared environment.

As discussed briefly in the Entity Service document, many of the functionalities associated with the dynamic fluid effort are individually relatively small. In particular, simulating, visualizing, and others are far too small to warrant allocation of a machine. This dilemma leads to two fundamental approaches. Many, or perhaps all, of these small dynamic fluid functionalities could be bundled into one large eclectic application, or each of these small functionalities could reside in separate applications.

Because development and maintenance would be unnecessarily complicated by the increased coupling and weaker encapsulation of the dynamic fluid functionalities if they were bundled into a single application, we chose to split the functionalities into separate applications. As discussed further below and in other parts of the documentation, we have found significant expressive power based on this decision.

It is important that a shared environment exists, and it would be ideal if applications were insulated from the duties of managing that shared environment. This insulation has been thematic in our approach. Better encapsulation of the environment, and the applications, not only simplifies development, but also helps to localize the impact of adding new capabilities in the future. Whether a dynamic fluid application or a more "standard" simulator, the software is dramatically easier to build and maintain with prudent use of abstraction and encapsulation.

5.1.2 Problem Statement

The requirement for the Fluid Service is a single application that can perform Dynamic Fluid I/O (input/output), maintain state for an arbitrary number of fluid attributes, and support several simultaneous client applications that require independent subsets of this fluid state information.

5.1.3 Solution

Several design goals are related to the Fluid Service, including:

- Decouple the application from the form of the data - This is imperative with respect to the design of a flexible architecture. It is not safe to assume that the data forms currently in use will necessarily be used in the future, nor is it safe to assume that new desired functionalities will necessarily fit any currently used form. In this case, the form referred to is the communications protocol (prototypical Fluid PDUs). With our implementation the client applications do not know, nor should they know, what the underlying fluid database looks like. The mindset is different. *No longer do applications think in terms of what they can get out of the database, but rather they think in terms of what they want from the database.*
- Find effective data abstractions for relaying fluid information - Decoupling the application from the form of the data requires the development of effective data abstractions that can be used as intermediaries.
- Support a small, but arbitrary number of applications - To provide for future demands, no artificial constraints should be placed on the number of client applications that can be running on a single machine at any given time. The principal constraints should be performance requirements and resource availability. If twenty lightweight applications could perform on a single machine, it should be possible to allocate all twenty of those to a single machine.
- Flexibility - The Fluid Service should be flexible enough to handle changes to the protocols (both between the service and the network, and between the service and the applications). Future requirements may require additional state information for the fluid.

5.2 Analysis

The cornerstone of our approach is the use of object oriented analysis and design techniques. The object oriented paradigm is characterized by an iterative, incremental approach in which both the understanding of the problem and the design of a solution for that problem evolve over time from the general towards the specific. It is important to obtain results early, and to use the knowledge gained to refine subsequent iterations.

As part of the iterative approach, we have chosen to enumerate and address high-risk areas as early as possible. In the early iterations, we want to face the issues that have the highest potential to cause radical changes in the design. In some sense, we want to build a reasonable skeleton before we try to flesh it out. Areas identified early as high-risk for the Fluid Service include:

- Active Fluid database - What sort of database representation will allow dynamic updates at runtime? Can the database make use of standard data

sources and still have enough expressive power to handle arbitrary resolution? Can the database be flexible enough to add unforeseen attributes as required?

- Identifying a data abstraction for fluid state information - What is the form of the data exchanged between the service and the client applications? This abstraction ultimately becomes the sandbox for experimental protocols. What are possible/reasonable/useful ways to handle this experimentation?

- Interprocess communications - What are possible/reasonable/useful ways to handle communication between the service and the client applications? What are the tradeoffs between performance and complexity?

5.3 Implementation

5.3.1 Problem Statement

Taking into account the stated design goals and the desire to address the high-risk areas early, a single application is needed to serve as the interface between the client applications and the shared environment for fluid state information.

5.3.2 User's Guide

5.3.2.1 The Structure of the Relationship

The Fluid Service as we've defined it runs in its own process and serves as the intermediary between the client applications and the network. There is one copy of the Fluid Service per physical machine. See Figure 25.

We have encapsulated the functionalities of communication and maintenance of Fluid state within the Fluid Service. We have also gained the capability to support a small, arbitrary number of applications on the same machine (currently a maximum of twenty-four client applications). The ripple effect is that the applications can be further decoupled from each other and more tightly encapsulated.

The Fluid Service provides a mechanism for applications to log on and off. Logging on or off can occur at any point before or during a running scenario. The details of the procedure are handled internally, so that the client application programmer does not need to be concerned with them.

5.3.2.2 Segregation of Responsibility Between the Fluid Ser-

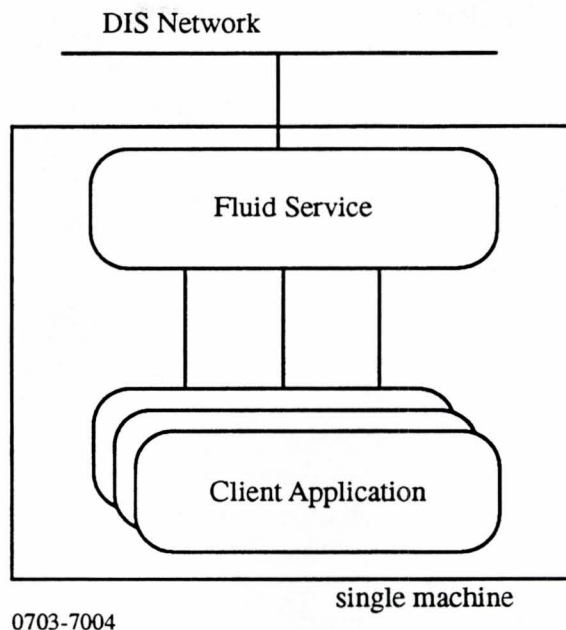


Figure 25. The logical relationship between the Fluid Service and the client applications on a single machine.

vice and the Client Applications

Responsibilities are divided between the Fluid Service and the client applications in a manner that reflects the goal of decoupling the applications from dealing with the network. The duties of the Fluid Service include:

- Communicate with other services via the network. (This includes being able to read and write prototypical PDUs.)
- Provide meaningful yet generic forms for the transfer of fluid related information between the service and the client applications.
- Provide useful and flexible mechanisms for the client application to obtain to the information contained within the service.
- It is safe for the client applications to assume that the “most current” fluid state information is available from the service.

Responsibilities of the client include:

- Properly make use of the mechanisms provided by the service.
- Translate between the generic form provided by the service and any specialized form needed by the client application. (If required, this conversion would be handled as in the Entity Service)

The mindset of the application should be to assume that the Fluid Service is "...a process that I can link up with at runtime to get the fluid state information that I need."

5.3.2.3 Addressing the High Risk Areas

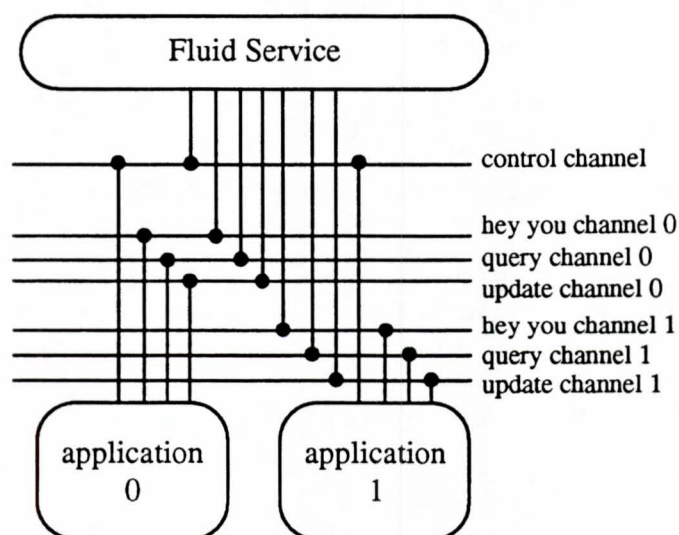
Developing a database that can represent the fluid in a flexible way is important. Furthermore, the ability to store multiple Fluid attributes is a key aspect when considering vehicle mobility. Refer to the Dynamic Fluid Database document for more information.

Identifying a data abstraction for fluid information is an area that deserves further scrutiny. For now, the structure of the information passed between the service and the applications is similar but not identical to the prototypical Fluid PDU.

Interprocess communication is another area of concern. Conceptually, the links between the service and the client applications can be thought of as communications "channels". These channels represent logical connections. See the figure below.

There is one "control" channel set up by the Fluid Service upon its creation that is intended as the means of communicating control, or management information (information that is outside the scope of the running scenario). This version's only control centers around the client applications logging on and off (i.e., establishing and breaking contact with the service).

There are also "hey you," "query" and "update" channels created for each application when it logs on to the service. The hey you channel carries fluid update information that has been received from the network or other client applications, the query channel is used to support fluid



0703-7006

Figure 26. Logical communication channels between the Fluid Service and the client applications.

attribute queries from clients, and the update channel carries fluid update from the client applications.

Currently, the physical communication between the Fluid Service and the client applications is accomplished using shared memory protocols with semaphores and boolean flags for controlling access to the shared memories. However, this is an implementation detail that should not be considered by the client application programmer. The physical implementation is subject to change in the future.

5.3.2.4 The Fluid Service - A Client Programmer's View

Several software classes comprise the functionality provided by the Fluid Service. These include classes for handling the network, the active database, and the shared memory interface. It is important to note that one of these classes, the `FluidServiceClientInterface`, does not appear within the Fluid Service. Instead, it is instantiated within the client application and serves as the client's end of the shared memory "wire." Basically, it insulates the application programmer from dealing with the shared memory and semaphore/boolean flag concerns. It is almost as if the application makes a function call (to the Fluid Service) to obtain the information needed.

5.3.2.4.1 Normal Operation of a Client Application

The normal operation of the client application interacting with the Fluid Service can be thought of as a client session. That is, the application "logs on" to the Fluid Service, receives fluid state information as needed, and then "logs off" when complete.

The process of logging on occurs in the constructor of the client interface. The client programmer simply instantiates the `FluidServiceClientInterface`. Similarly, logging off is performed in the destructor of the client interface. An obvious physical requirement is that the Fluid Service process must be running prior to running any client applications.

Logging on is handled via a "well known" shared memory. That is, when the service starts up it allocates its control shared memory using a known shared memory key. Subsequently, when an application starts up, it attaches to the shared memory using the same known key and logs on to the service. This well known key is a configuration item. Logon structures within the control shared memory allow for application control information to flow between the service and the application. These same structures provide the detailed means for applications to log off when they are done.

Getting fluid state information from the Fluid Service is designed to be effortless to the client application programmer as possible. The client asserts the Fluid Service to query for state information, then a period of time for the service to process the request, and then reads the fluid state information. This process is diagrammed in Figure 28, and a code fragment is provided in Figure 27.

Note that the assertion and the wait are separated into two calls by the client so that some parallelism can be achieved. If the assert is followed immediately by the wait, then the client program will

somewhere earlier...

```
client = new FluidServiceClientInterface("test", Hz, wellKnownKey,  
channelsMask, maxFluids, errorCode);
```

somewhere in the main loop...

```
client->assertAwaitingHeyYou();           // alert service  
    // other processing can be done in here...  
client->waitForHeyYou();                   // data ready?  
while (client->readHeyYou(&em)) {          // read one entity  
    // do whatever...  
}
```

Figure 27. A code fragment showing the state transfer interaction between the client application and the Fluid Service.

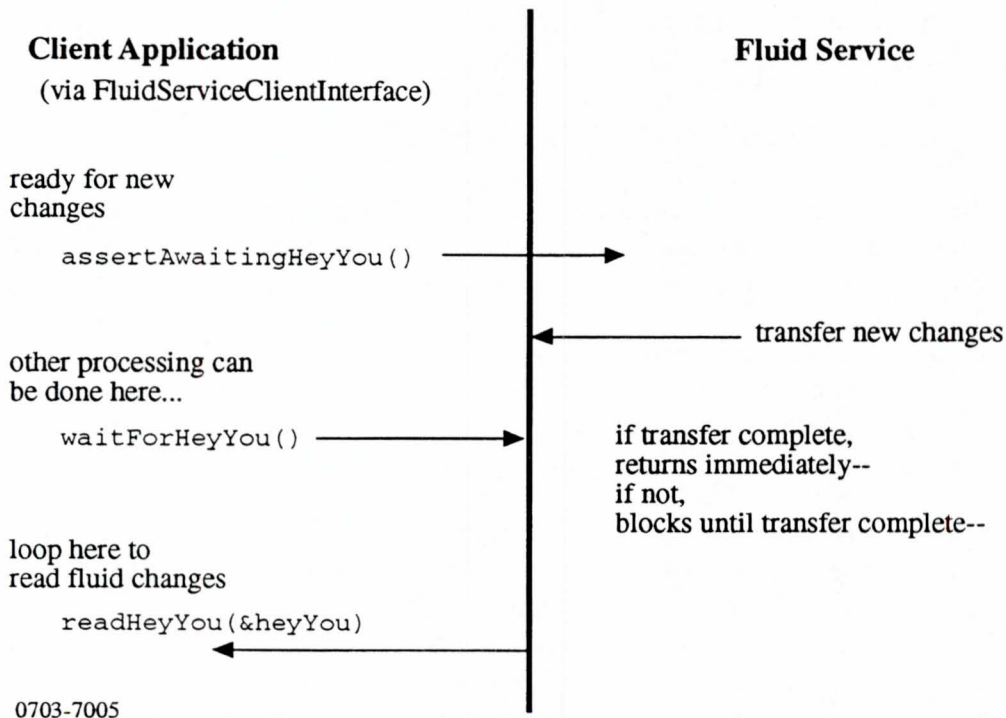


Figure 28. A schematic representation of the "Fluid changed" transfer interaction between the client application and the Fluid Service.

block until the service has processed the query. If, on the other hand, there is useful work that the client application can be doing in the meantime, then the assert and the wait can straddle that work. If the Fluid Service has completed the processing of the query prior to the wait call, then the wait returns immediately.

For situations where a particular client application wants to make a change to a fluid, there is a companion “write” that is the reverse of the “read” described above. The basic requirement is that when the application wants to update the fluid, it writes that information to the Fluid Service.

Note that the client application does not need to be concerned with when or how fluid PDUs are broadcast to the network-- that is the responsibility of the Fluid Service. It is only required that the application inform the service of desired changes in fluid state.

5.3.2.4.2 The FluidServiceClientInterface Class

The FluidServiceClientInterface class represents the client application’s end of the shared memory connection with the Fluid Service. The public interface to this class is described below.

```
class FluidServiceClientInterface
{
    public:
        // The constructor for the client interface is called with
        // parameters for the name of the application, the desired
        // frame rate of the application, the “well-known” shared
        // memory key for logging on to the Entity Service, a
        // bit-mask (currently, the only available value is
        // FS_WANT_HEY_YOUS) describing which messages the
        // application will ask for, a maximum number of bodies
        // of fluids that will be queried or updated at one time,
        // and a parameter to use for notifying the application
        // about errors during the logon process. The client
        // interface will, during the constructor, attempt to log
        // on to the Fluid Service. Failure to log on to the
        // service will result in a fatal error.
        // (Make sure the service is running first)
        FluidServiceClientInterface( char *appName,
                                    int appFrameRate,
                                    key_t controlShmkey,
                                    int channelsMask,
                                    int maxFluids,
                                    int *errorCode );

        // The destructor is responsible for logging off of the
        // Fluid Service and cleaning up.
        ~FluidServiceClientInterface();

        // The next three routines provide reading capability to
        // the client application for hey you (a piece of Fluid
        // changed) information. The reading functions should
        // only be called if FS_WANT_HEY_YOUS was specified in
        // the channelsMask of the constructor.
        // -----
        // The assertAwaitingHeyYou call is the signal to the
        // Fluid Service that the client application needs to
        // get the new notifications of changed Fluid.
        int      assertAwaitingHeyYou();
}
```

```

// The waitForHeyYou call delays the read of hey you
// information until the service has finished processing
// the request.
int    waitForHeyYou();

// The readHeyYou call is looped on by the client
// application to get the new notifications of changed
// Fluid. The readHeyYou call returns NULL when all
// notifications have been read.
int    readHeyYou( AttributeMsg *msg );

// The next three routines provide querying capability to
// the client application for the Fluid database.
// -----
// The assertAwaitingQuery call is the signal to the
// Fluid Service that the client application needs to
// get (query) a piece of the Fluid database. The
// parameter tells the Fluid Service which portion of
// the database to query and at what resolution.
int    assertAwaitingQuery( AttributeMsg *msg );

// The waitForQuery call delays the read of the returned
// query information until the service has finished
// processing the request.
int    waitForQuery();

// The readQuery call places the returned data from the
// query in the parameter.
int    readQuery( AttributeMsg *msg );

// The writeUpdate call is used by a client application
// to perform an update to the Fluid database.
// Note that this call is standalone, no assertion is
// necessary.
int    writeUpdate( AttributeMsg *msg);
};

```

5.3.2.4.3 The Structures Used to Communicate with the FluidService-ClientInterface

The structures used in the read and write calls to the FluidServiceClientInterface are detailed below (see the Fluid DTR document for information about the FList structure). Note that a further enhancement of the Fluid Service could be to remove knowledge of the "FList" structure from the clients and simply pass a list of vertices for each body of water.

```

typedef struct
{
    u_char    fluid;
    FList     *flEndHead;
    FList     *flEndTail;
} FSMsgData;

typedef struct

```

```

{
    FSPDUHeader    header;
    u_char          numberOfBlocks;
    FSMsgData       block[MAX_FLUID_BLOCKS];    // A patch of fluid
} FluidMsg;

```

5.3.2.4.4 The Configuration File

The Fluid Service requires a configuration file in order to operate. The file has eight variables that must be set:

- **fservHz n**
where n is the number of times per second that the Service should process entities and miscellaneous information.
- **dfdbConfigFilename n**
where n is the filename of the configuration file for the Dynamic Fluid Database (DFDB)
- **maxClients n**
where n is the maximum number of clients that this service should provide for.
- **controlShmkey n**
where n is the “well-known” shared memory key for the control channel.
- **nextShmkey n**
where n is the first shared memory key to use for client channels (hey you, query and update)
- **deltaShmkey n**
where n is the value to add to nextShmkey for each channel after each client logs on (i.e. the difference between successive channel shared memory keys).
- **netConfig $t k$**
where t is one of “server”, “client”, “relay” or “distributed” that defines in what mode the Service will operate, and k is the key of the link that goes toward a central server (note that k is only used in a Service running in “relay” mode). “Modifying the FS Configuration” on page 249 discusses the different network configurations in detail.
- **link $k i p$**
where k is a unique key assigned to this link, i is the IP address of machine to connect with, and p is the port number to use for the connection.

A simple configuration file might look like:

```

FSERV_CONFIG_FILE

fservHz      40.0

dfdbConfigFilename  dfdb.cfg

maxClients   10

controlShmkey 0x00006000
nextShmkey    0x00010000
deltaShmkey   0x00000010

```



```
netConfig    distributed

link    0    127.0.0.1    5000
```

Note that the first line (FSERV_CONFIG_FILE) is required. Once configuration file is created, run the Fluid Service by placing the name of the configuration file as a command-line parameter to the Fluid Service. For example, "fluidServ fserv.cfg" will run the Fluid Service using a configuration file named fserv.cfg.

5.3.3 Software Design

Figure 29 shows the Booch diagram of classes used in the Fluid Service. To better facilitate the iterative design process as well as maintenance, the classes involved were abstracted into a hierarchy so that common implementation details exist in one location. In fact, all three services (Entity, Terrain and Fluid) are based on the same core classes. Furthermore, the Fluid Service was built based on the Terrain Service in very little time due to the abstraction developed earlier.

5.3.4 Programmer's Guide

A guide to the programmer of the Fluid Service is intuitively divided into two sections: one for a "client application programmer" that contains the detail of the client interface, and one for a "service programmer" that contains details on semaphores, shared memory, network interface, "channels," and structure of the service. However, it is important to note that the Fluid Service uses Unicast network connections because the size of the prototype PDU is too large for broadcast methods.

5.3.4.1 Service Programmers

In order to spend more time discussing the structure of the service and the "channels," basic knowledge of semaphores, shared memory and network interfacing is assumed. Since the service and the client applications exist as separate processes, they communicate through common structures in shared memory. Semaphores are used to control access to these common data structures including the control structure (for logging on and off) as well as the state and miscellaneous channels for each client application. The network interface is used to communicate with other machines (reading and writing of PDUs).

In order to provide the capability of communication between the service and the client applications, a number of logical "channels" were developed (as mentioned earlier). Each channel provides a pathway for one type of data. For example, the Fluid Service uses four types of channels: one for control information, one for hey yous (fluid change notifications), one for fluid queries and one for fluid updates. The structure of each channel appears as follows:

```
typedef struct
{
```

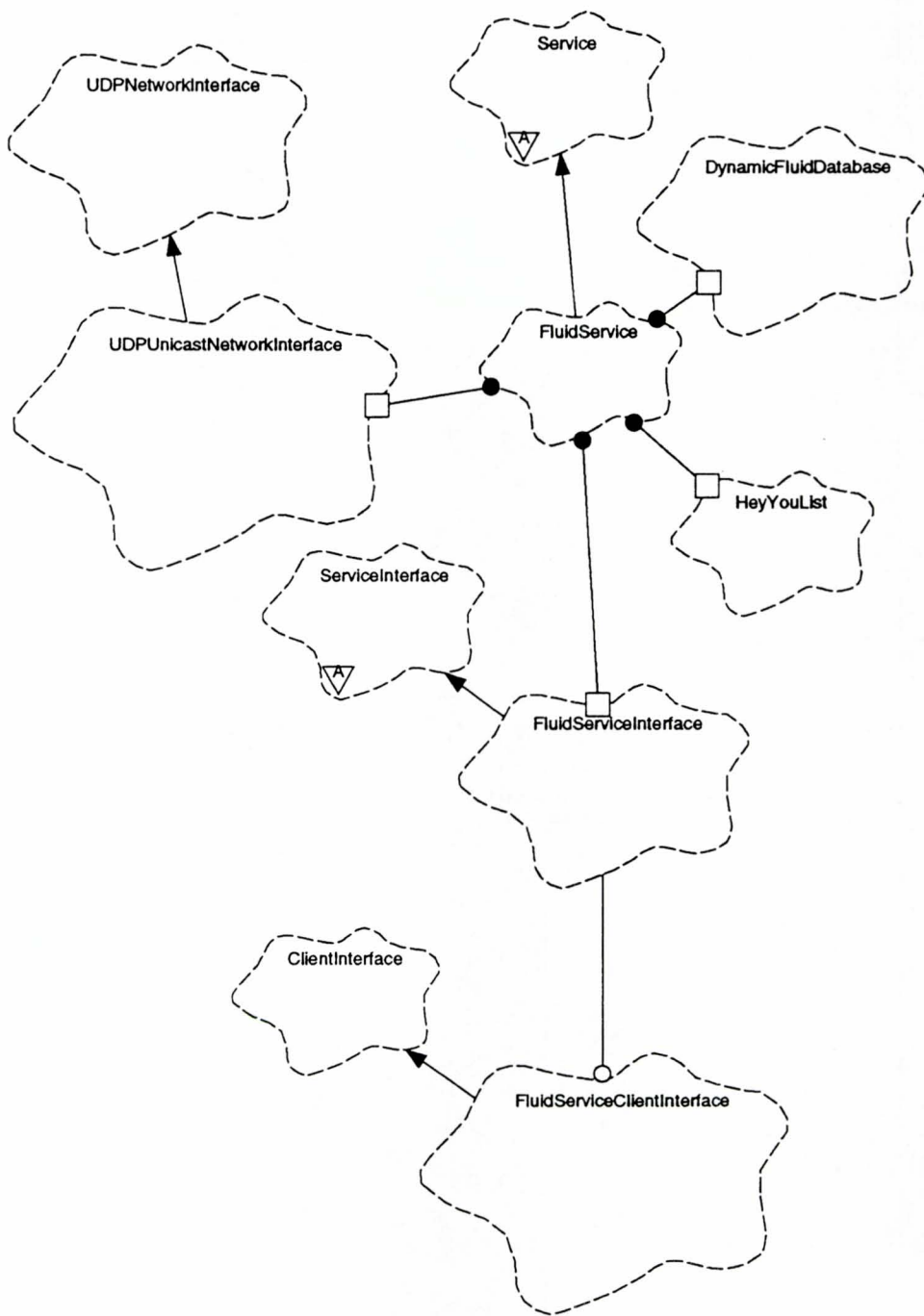


Figure 29. Booch diagram of classes for the Fluid Service

```

    int         inUse;
    pid_t       appPid;
    char        appName[80];
    int         appFrameRate;
    int         wantsToLogon;
    int         wantsToLogoff;
    int         appIndex;
    int         channelsMask;
    int         variableSize;
    int         loggedOn;
    key_t       accessShmkey;
    key_t       channelShmkeys[MAX_CHANNELS];
    key_t       variableBlockShmkeys[MAX_CHANNELS];
} LogonStruct;

// Control Channel
typedef struct
{
    int         numberToLogon;
    int         numberToLogoff;
    int         maxApplications;
    LogonStruct logon[MAX_APPLICATIONS];
} ServiceControlChannel;

// Hey You Channel
typedef struct
{
    AttributeGram inbound[FS_MAX_INBOUND_HEY_YOUS];
    int         numInbound;
    int         nextInboundRead;
    int         nextInboundWrite;
} FSHeyYouChannel;

// Query Channel
typedef struct
{
    AttributeGram inbound[FS_MAX_INBOUND_QUERIES];
    int         numInbound;
} FSQueryChannel;

// Update Channel
typedef struct
{
    AttributeGram outbound[FS_MAX_OUTBOUND_UPDATES];
    int         numOutbound;
} FSUpdateChannel;

```

Note that the HeyYou and Update channels contain only one list each since they are unidirectional channels. HeyYous travel towards the client applications since they notify them of fluid changes. Updates travel towards the Fluid Service since they tell the service of an update that the client application wants to make.

5.3.4.2 Client Programmers

As mentioned earlier, the client interface provides a means of information hiding. The client applications should not have to deal with issues such as shared memory communication, semaphores, etc. This is all hidden in the client interface. Furthermore, it is important to note that the client interface can also contain any information specific to the individual client. For example, coordinate conversions of the fluid updates could be done within the FluidServiceClientInterface if needed or wanted in the future. Therefore, by the time the service receives the data, it will have been already transformed into geocentric (as required by DIS).

Dynamic Terrain Resources - Shared Environment Clients

This section describes the Dynamic Terrain Resources (DTRs) which are clients of the Shared Environment discussed in the previous section "Dynamic Terrain Shared Environment" on page 1.

Programs that make use of the Shared Environment are collectively known as Client Applications. Client Applications are responsible for simulating some element of the shared virtual space. This element can be either a traditional entity or the simulation of some part of the environment. Client Applications typically run on a single physical machine; however, in some cases they may be composed of several separate processes.

The broad category of Client Applications is further broken down into DTRs and other client applications. DTRs typically implement a single functionality associated with changing the terrain. Collectively, the available DTRs represent the logical terrain player for any given scenario. Implementing these functionalities as separate programs greatly simplifies the creation and maintenance of the software, as well as providing for a high degree of flexibility as to how the logical terrain player is configured for any particular scenario. For instance, if it is known that a particular scenario does not require any craters, then simply do not run the craterDTR.

Various DTRs described in the following sections include the Soil DTR, Fluid DTR, Crater DTR, Track DTR, Thermal DTR, and Minefield DTR.

6.0 Soil DTR

6.1 Introduction to Soil DTR

6.1.1 Background

Physically-based modeling is a growing area of computer graphics research. A good deal of work has been done toward physically-based models of objects such as rigid and nonrigid bodies, hydraulic surfaces, and natural terrain. Dynamic soil models are required in animations and real-time interactive simulations in which changes of natural terrain are involved. Terrain activities on a dynamic terrain include digging ditches, building berms, generating craters, and more complicated military T-push maneuvers in which two bulldozers operate in tandem to create a berm.¹

6.1.2 Problem Statement

Soil deformations should behave in a realistic manner in response to the external stimuli of dynamic terrain activities. Kinematic soil models are commonly implemented in soil animation; however, a physically-based model is a mathematical representation of its behavior that incorporates principles of Newtonian physics. The physically-based model includes soil parameters that allow it to mimic various soil types such as dry sand, sandy loam, and loam. The algorithm must be efficient to run in an interactive rate simulation.

6.1.3 Solution

Analytic methods, based on soil properties and Newtonian physics, were developed to model soil slippage. The soil slippage model determines if a given soil configuration is in static equilibrium by calculating forces which drive a portion of the soil to slide if the configuration is unstable. Meanwhile, it operates under the constraint of volume conservation to simulate excavating activities such as digging, cutting, piling, carrying, or dumping soil. Numerical algorithms with linear time and space complexities were implemented to meet the requirement of real-time computer simulation.

6.1.4 Constraints/Assumptions

Since soil is a complex material and modeling manipulations of arbitrarily complex soil configurations is either intractable or costly, the soil model was developed under the following assumptions:

- Both homogeneous and heterogeneous soil parameters are supported. The original soil model was developed for use with homogeneous soils only; however, the model was

1. In this maneuver, two bulldozers are positioned perpendicular to each other and dig a straight-line ditch with a running anti-tank berm on the opposing-force side of the ditch. The name T-push refers to the "T"-shape cuts made during the maneuver.

extended to accomodate heterogenous soil parameters within a terrain database.¹

- Conditions such as seepage, pore pressure, existence of tension cracks, and deformation resulting from permanent atomic dislocation are not considered.

6.2 Analysis

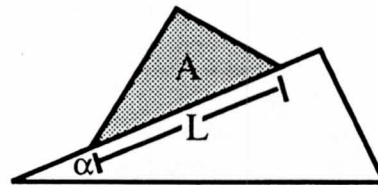
The discussion of soil models requires some understanding of soil properties. In this section, some concepts which are borrowed directly from civil engineering are introduced.²

6.2.1 Preliminaries

The soil's shear strength per unit area is the resistance to deformation of soil particles along surfaces of rupture. Deformation occurs by continuous sheer displacement. It may be attributed to three basic components:

- frictional resistance to sliding among soil particles
- cohesion and adhesion among soil particles³
- interlocking of solid particles to resist deformation.

The shear stress, on the other hand, is the force per unit area experienced by a slope that pushes the mass to move along the failure plane. The combined effects of gravity and water are the primary influences on shear stress. It may also be influenced by some natural phenomena such as chemical actions, earthquakes, or wind.



0703-6982

Figure 30. Failure Plane

The shear strength force and stress force, denoted by s and τ respectively, are defined as the shear strength and stress multiplied by the total area. The measure of s and τ can be determined from the Mohr-Coulomb theory (refer to the above figure):

$$s = c L + W \cos(\alpha) \tan(\phi) \quad (1)$$

$$\tau = W \sin(\alpha) \quad (2)$$

where

L = length of the failure plane,

-
1. The extension to the original soil model is simply a modification to accomodate a database with heterogeneous soil parameters. It did not undergo the detailed analysis of the homogeneous soil model.
 2. Most of this section is an excerpt from [36].
 3. Cohesion is molecular attraction among similar particles. Adhesion is a molecular attraction among unlike particles.

α = degree of natural slope,
 $W = \gamma A$ = weight of soil above the failure plane,
 A = area of soil above failure plane,
 c = cohesion,
 ϕ = angle of internal friction (i.e. a measure of the friction among soil particles),

and

γ = unit weight .

Soil is a complex material. It may be influenced by changes in the moisture content, pore pressures, structural disturbance, fluctuation in the ground water table, underground water movements, stress history, time, chemical action, or environmental conditions. Predicting the changes of complex configurations is either intractable or costly. However, for many interactive applications, speed and realistic appearance are more important than accuracy. Hence, the constraints mentioned in "Constraints/Assumptions" on page 81 were imposed.

6.2.2 Static Equilibrium and Restoring Force

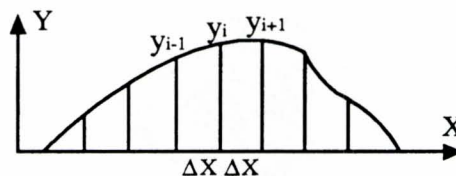
In this section, methods are developed to determine whether or not a given configuration is stable. The critical angle above which sliding occurs is calculated and the force that pushes the soil mass along the failure plane is quantified.

6.2.2.1 Stability

The stability of a given soil configuration is determined by the factor of safety, F , of a potential failure surface. From the Mohr-Coulomb theory, F is defined as a ratio between the strength force and the stress force:

$$F = \frac{s}{\tau} = \frac{cL + W \cos(\alpha) \tan(\phi)}{W \sin(\alpha)} \quad (3)$$

When F is greater than 1, the configuration is said to be in a state of equilibrium. Otherwise, failure is imminent. To analyze the factor of safety, the given soil mass is divided into n slices with equal width Δx :

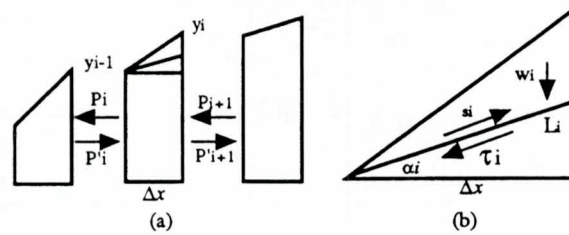


0703-6983

Figure 31. Soil mass division into n slices

The calculation of the factor of safety of each slice can be done individually. The following free

body diagram shows the forces applied on slice i :



0703-6984

Figure 32. Free body diagram for slice i

In Figure 32(a), the P 's are forces exerted between slices. They are pairwise equal and in opposite directions; thus, they can be cancelled. Therefore at any time, t , sliding can occur only in the top triangular area of a slice. Figure 32(b) shows forces acting on this area, where strength and stress forces are given by equation (1) and equation (2) with L , W , and α replaced by L_i , W_i , and α_i respectively.

The next step involves determination of a failure angle, α_i , to see if the soil mass above it will slide. If the failure angle exists, then the net force exerted on the failure plane is calculated. Note that L_i and W_i can be expressed in terms of α_i . Replacing L_i and W_i with functions of α_i results in:

$$F(\alpha_i) = [2c + \gamma \tan(\phi) [h \cos(\alpha_i) - \Delta x \sin(\alpha_i)] \cos(\alpha_i)] / [\gamma (h \cos(\alpha_i) - \Delta x \sin(\alpha_i)) \sin(\alpha_i)] \quad (4)$$

where $h = y_i - y_{i-1}$ is the height of the triangle in Figure 32(b). For any angle $\alpha_i > \tan^{-1}(h/\Delta x)$, function $F(\alpha_i)$ makes no physical sense. In the range of $[0, \tan^{-1}(h/\Delta x)]$, $F(\alpha_i)$ reaches its minimum when the first derivative of $F(\alpha_i)$, with respect to α_i , is equal to 0. That is

$$dF/d\alpha = (1/\tau^2) [A \cos(2\alpha_i) + B \sin(2\alpha_i) + C] = 0 \quad (5)$$

where

$$A = (\gamma^2/2) \tan(\phi)(\Delta x^2 - h^2) - 2\gamma ch \quad (6)$$

$$B = \gamma^2 h \Delta x \tan(\phi) + 2\gamma c \Delta x \quad (7)$$

and

$$C = -(\gamma^2/2) \tan(\phi)(\Delta x^2 + h^2). \quad (8)$$

Solving equation (5) yields four angles¹. We can choose the one which satisfies $0 \leq \alpha_i \leq \tan^{-1}(h/\Delta x)$ in equation (5) to calculate the factor of safety F . The given configuration is statically stable if $F > 1$. Otherwise sliding is inevitable.

1. see [7]

6.2.2.2 Critical Slope Angle

Consider the case where $F < 1$ for a given configuration. In the range of $[0, \tan^{-1}(h/\Delta x)]$, there are at most two angles, say β_1 and β_2 , such that $F(\beta_1) = F(\beta_2) = 1$. The angle $\beta_0 = \min(\beta_1, \beta_2)$ is said to be the critical-slope angle of the configuration. Above this angle impending slip occurs. β_1 and β_2 can be obtained by solving the equation for α :

$$F(\alpha) = [(2c + \gamma \tan(\phi)) [h \cos(\alpha) - \Delta x \sin(\alpha)] \cos(\alpha)] / [\gamma [h \cos(\alpha) - \Delta x \sin(\alpha)] \sin(\alpha)] = 1 \quad (9)$$

where all symbols are as explained earlier.¹

6.2.2.3 Restoring Force

Figure (4a) shows a soil configuration with β_0 as the critical-slope angle. The force that pushes the mass in the triangle along the edge gh_0 can be computed as follows. First the line segment h_0h_n is divided into n small segments with equal length Δh . Figure (4b) shows the free body diagram of the i -th dovetail indicated by the shaded area in (4a).

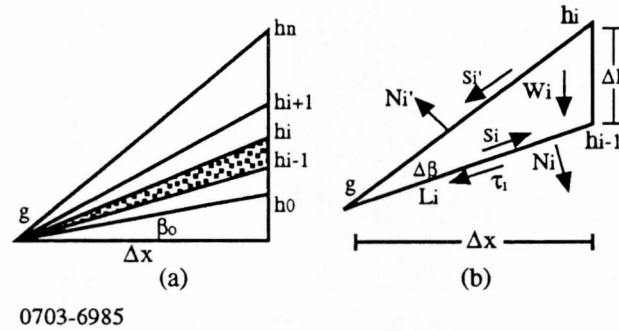


Figure 33. Analyzing the restoring force

Analysis of the forces exerted on the dovetail follows. The weight W_i can be decomposed into two forces, namely N_i and τ_i , which are normal and parallel to the edge L_i respectively. s_i is the strength force resisting the sliding motion, s_i' the opponent force generated by strength force s_{i+1} , and N_i' the force supporting the dovetail above it. The net force f_i applied on dovetail- i is therefore given by a vectorial summation:

$$f_i = N_i + \tau_i + s_i + s_i' + N_i' \quad (10)$$

The total net force f acting on the whole triangle area is the summation of f_i 's, $1 \leq i \leq n$, i.e.

$$f = \sum_{i=1}^n (N_i + \tau_i + s_i + s_i' + N_i') = \sum_{i=2}^n \tau_i \quad (11)$$

since $\tau_1 = s_1$ (due to $F(\beta_0) = 1$), $N_n' = 0$, $s_n' = 0$, $N_i' = N_{i+1}$ and $s_i' = -s_{i+1}$ for $1 \leq i \leq n-1$. Based on equa-

1. The solution to equation (9) is derived in [7].

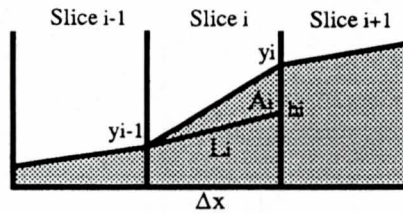
tion (11) and Figure ,and by letting Δh tend to zero the following derivation results¹

$$f = (\gamma \Delta x^2 / 4) [\ln((h_n^2 + \Delta x^2) / (h_0^2 + \Delta x^2))] \cos(\beta_0) + (\gamma \Delta x / 2) (h_n - h_0 - \Delta x(\beta_n - \beta_0)) \sin(\beta_0) \quad (12)$$

where $\beta_n = \tan^{-1}(h_n / \Delta x)$ and $\beta_0 = \tan^{-1}(h_0 / \Delta x)$. The total force on the top triangle area of each slice can be quantified by equation (12).

6.2.2.4 Volume Conservation

Recall that a given configuration is divided into n slices as depicted in the figure below. The i -th slice, $1 \leq i \leq n$, can be conveniently thought of as a container holding an amount of soil whose quantity is given by $(y_i + y_{i-1})\Delta x / 2$.



0703-6986

Figure 34. Consideration of slices as containers

Consider a small change, denoted by ΔW_i , of the mass W_i in slice _{i} . Since $W_i = (y_i + y_{i-1})\gamma \Delta x / 2$,

$$\Delta W_i = (y_i + \Delta y_i + y_{i-1} + \Delta y_{i-1})\gamma \Delta x / 2 - (y_i + y_{i-1})\gamma \Delta x / 2 = (\Delta y_i + \Delta y_{i-1})\gamma \Delta x / 2 \quad (13)$$

On the other hand, assume that there is a force f_i exerted on the triangle area A_i at the top of slice _{i} , which is parallel to the edge L_i . Due to f_i , A_i tends to move along the direction of f_i at a velocity v_i . The rate of the "flow" of mass of A_i through slice i can be computed by $\gamma A_i v_i / \Delta x$. Thus, the "mass throughput" of slice _{i} can be quantified by $\gamma A_i v_i \Delta t / \Delta x$, where Δt is a unit of time. Similarly, the mass throughput of slice _{$i+1$} is given by $\gamma A_{i+1} v_{i+1} \Delta t / \Delta x$.

From the principle of volume conservation, the change of soil quantity in slice _{i} is the amount of soil which goes out, minus the amount of soil which goes in. It can be expressed by:

$$\Delta W_i = (\gamma A_i / \Delta x) v_i \Delta t - (\gamma A_{i+1} / \Delta x v_{i+1}) \Delta t \quad (14)$$

where $A_i = (y_i - h_i)\Delta x / 2$. Putting equation (13) and equation (14) together and rearranging,

$$(\Delta y_i / \Delta t) + (\Delta y_{i-1} / \Delta t) = (1 / \Delta x) [(y_i - h_i)v_i - (y_{i+1} - h_{i+1})v_{i+1}]. \quad (15)$$

Now let Δt tend to 0. It follows that

1. derivation in [7]

$$(dy_i/dt) + (dy_{i-1}/dt) = (1/\Delta x) [(y_i - h_i)v_i - (y_{i+1} - h_{i+1})v_{i+1}] \quad (16)$$

Recall that equation (12) gives us a formula to compute force f_i . From Newton's second law,

$$f_i = \gamma A_i (dv_i/dt) = (\gamma \Delta x / 2) (y_i - h_i) (dv_i/dt) \quad (17)$$

Rearranging, results in both

$$(dv_i/dt) = 2f_i / [\gamma \Delta x (y_i - h_i)] \quad (18)$$

and

$$v_i = \frac{2}{\gamma \Delta x} \int \frac{f_i}{y_i - h_i} dt \quad (19)$$

Taking the second derivative of equation (16) with respect to t and substituting equation (18) and equation (19) into the resulting formula yields

$$\frac{d^2 y_i}{dt^2} + \frac{d^2 y_{i-1}}{dt^2} = \frac{2}{\gamma \Delta x} \left[\frac{dy_i - dh_i}{dt} \int \frac{f_i}{y_i - h_i} dt + f_i - \frac{dy_{i+1} - dh_{i+1}}{dt} \int \frac{f_{i+1}}{y_{i+1} - h_{i+1}} dt + f_{i+1} \right] \quad (20)$$

Note that h_i and f_i can be denoted as functions of y_{i-1} and y_i , i.e. $h_i = h(y_{i-1}, y_i)$ and $f_i = f(y_{i-1}, y_i)$, since they can be determined based only on y_{i-1} and y_i if Δx and other soil properties are fixed.

Hence equation (20) has three variables, namely y_{i-1} , y_i , y_{i+1} . Suppose that the given soil configuration is divided into n slices. This yields $n+1$ unknowns, y_0, y_1, \dots, y_n , and $n+1$ ordinary differential equations involving y_i 's, their time derivatives and integrals. Solving these equations, the solution for the soil behavior which satisfies both the soil dynamics and the volume conservation is obtained.

6.2.2.5 Numerical Solution

In this section, equation (20) is linearized for both purposes of simplification and discretization. Assume that, at any instance of time, t_m , velocity, v_i , of the mass on the top of slice i is represented by $v_i(t_m)$, the value of y_i is represented by $y_i(t_m)$, the rate of the change of y_i is represented by $y_i'(t_m) = dy_i(t_m)/dt$. Then, at the very next time instance t_{m+1} , the force $f_i = f_i(y_{i-1}(t_m), y_i(t_m))$ can be computed by equation (12) according to the value of y_{i-1} and y_i from the previous step. If the Euler integration algorithm is used, the velocity v_i at the time t_{m+1} can be computed by:

$$v_i(t_{m+1}) = v_i(t_m) + [(f_i(y_{i-1}(t_m), y_i(t_m)))/W_i] \Delta t \quad (21)$$

where Δt is the integration step size. Similarly $v_{i+1}(t_{m+1})$ is calculated. It follows that, from equation (17)

$$y_i'(t_{m+1}) + y_{i-1}'(t_{m+1}) = (1/\Delta x) [(y_i(t_m) - h(y_{i-1}(t_m), y_i(t_m)))v_i(t_{m+1}) - (y_{i+1}(t_m) - h(y_i(t_m), y_{i+1}(t_m)))v_{i+1}(t_{m+1})] \quad (22)$$

Since at the time instance t_{m+1} , all items on the right hand side are knowns, either from the pre-

vious step of the simulation or from the calculations of $v_i(t_{m+1})$ and $v_{i+1}(t_{m+1})$, it can be treated as a constant, namely C_i . Now there are $n+1$ equations in the following format:

$$y_0'(t_{m+1}) = C_0$$

$$y_1'(t_{m+1}) + y_0'(t_{m+1}) = C_1$$

.....

$$y_n'(t_{m+1}) + y_{n-1}'(t_{m+1}) = C_n$$

Solving the above for $y_i'(t_{m+1})$, $i=0, 1, \dots, n$, the Euler method can be used again to determine the new values for each y_i :

$$y_i(t_{m+1}) = y_i(t_m) + y_i'(t_{m+1})\Delta t \quad (23)$$

Algorithm 1 describes the procedure of the numerical solution, in which each step of the algorithm takes linear time to execute. Thus the time complexity of the algorithm is $O(n)$ where n is the number of elevation posts in a given configuration. The space required to store forces, velocities, and heights of posts is also proportional to n .

6.2.2.5.1 Algorithm 1

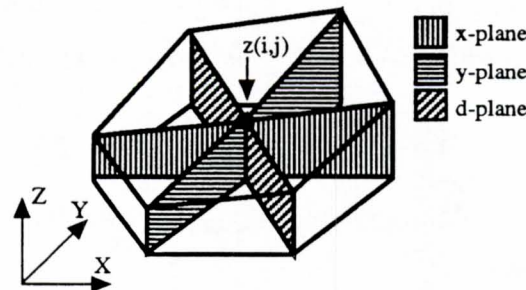
At any time t_{m+1} of simulation, perform the following:

1. For each post y_i , calculate its mass velocity $v_i(t_{m+1})$ by using equation (21).
2. For y_i , compute the right hand side of equation (22).
3. Use forward substitution to solve the set of $n+1$ equations for $y_i'(t_{m+1})$, $i=0, 1, \dots, n$.
4. Use Euler integration to determine new value for each $y_i(t_{m+1})$.

6.2.3 Extension to 3-D

In going to 3-d soil dynamics, some essential concepts and results from the discussion on 2-d are borrowed. First, a given soil configuration is partitioned into small prisms. The values of elevation posts (i.e. vertices) of each prism are evolved by an approximation procedure as follows.

Consider in the following figure, the post $z(i,j)$ chosen arbitrarily:



0703-6987

Figure 35. An approximation of the 3-d configuration

$z(i, j)$ is surrounded by six prisms. At any time instance t , those prisms are the only ones that

affect the height of $z(i,j)$. The effect caused by those prisms can be approximated by considering forces exerted on three planes, namely the x-plane, y-plane, and d-plane. They are indicated by different types of shaded areas in Fig. 6. Thus the 3-D problem is reduced to a 2-D problem. The finer the partitioning, the smaller the base triangles, and the more accurate the approximation.

Assume that, at any time t_m , the height of post $z(i,j)$ is represented by $z_{ij}(t_m)$, and the rate of change of $z(i,j)$ is represented by $z'_{ij}(t_m)$. Since $z'_{ij}(t_m)$ is affected by forces from 3 planes, it can be expressed as a summation of three terms:

$$z'_{ij}(t_m) = zx'_{ij}(t_m) + zy'_{ij}(t_m) + zd'_{ij}(t_m) \quad (24)$$

where $zx'_{ij}(t_m)$, $zy'_{ij}(t_m)$ and $zd'_{ij}(t_m)$, are rates of changes of $z'_{ij}(t_m)$ caused by forces exerted on the x-plane, y-plane and d-plane respectively.

During a simulation, each time slice Δt is divided into two substeps Δt_1 and Δt_2 . In Δt_1 , equation (12) is used to compute forces exerted on three different planes. Then $zx'_{ij}(t_{m+1})$, $zy'_{ij}(t_{m+1})$ and $zd'_{ij}(t_{m+1})$ can be obtained by solving the set of $n+1$ equations. In step Δt_2 , Euler integration is used to determine new values for each $z_{ij}(t_{m+1})$:

$$z_{ij}(t_{m+1}) = z_{ij}(t_m) + [zx'_{ij}(t_m) + zy'_{ij}(t_m) + zd'_{ij}(t_m)]\Delta t \quad (25)$$

For Δt_1 and Δt_2 of each iteration in the simulation, the 2-d computational problem is split into 3 terms: x-plane scan, y-plane scan and d-plane scan. Each scan has two phases corresponding to two time substeps. A scan on any plane involves calculations of forces exerted on that plane, rates of changes of $z(i,j)$ caused by the forces, new height of each post, etc. Computations for each scan in a time substep are independent of scans on the other planes in the same substep, and therefore can be performed either sequentially or in parallel. It is important to notice that, in the same time substep, scans in different orders (x-scan then y-scan then d-scan, or y-scan then x-scan then d-scan, etc.) will have the same effect.¹

The 3-d algorithm can be briefly described as follows: Each iteration of simulation is divided into two phases. Steps (1)-(3) of Algorithm 1 are performed first for each scan. Then step (4) is applied for each scan to calculate new values of posts. Both time and space complexity of the 3-d algorithm remain linear in the number of posts.

6.3 Implementation

6.3.1 Problem Statement

The soil model was the initial work which provided the momentum that became the Dynamic Terrain project. The original "C" code for the soil model was revised to a C++ class which is referred

1. The reasoning is discussed in [7].

to as the SoilClass. Terrain Service evolved after the soil model as a result of several DT architecture studies. This prompted the creation of the SoilDTR which is a client to the Terrain Service via the Terrain Service Client Interface (Section 4.3.2.4.2 The TerrainServiceClientInterface Class). This allows the Soil DTR to make changes to terrain elevations in a given soil patch. Upon return of the patch to Terrain Services, it is integrated into the global terrain. Aside from terrain elevation, the soil parameters of cohesion, density, and internal angle of friction are also obtained from the multilayered database.

6.3.2 User's Guide

Two implementations are possible with the soil model -- alone and in conjunction with Terrain Service. The Soil DTR provides an interface to Terrain Service that allows for query of terrain attributes and modification of the elevation attribute.

Alternately, the soil model can be used by itself. In a stand-alone simulation of a bulldozer, the SoilClass allows the implementor to slump the soil during excavation. The implementor simply calls the slumpSoil public method.

6.3.2.1 Configuration File

The Soil DTR has no configuration file; however it depends on values set in the DTDB configuration file (Section 1.3.2.3 Configuration). The DTDB configuration file must contain information for elevation, cohesion, density, and internal angle of friction attribute layers for the heterogeneous soil methods to work properly. An excerpt of a sample file follows.

```
# For layers in the terrain database, layer 0 is the elevation data
number_of_attributes      4

# the main "surface" is elevation (the ground's geometry)
# this is a paired entry (data_file, surface_rep) since
# elevation is always attribute 0
elevation_data_file       flat.data
elevation_surface_rep     uniform_nonrational_bilinear_Bspline

# the following entries are 4-tuples (attribute_key,
# attribute_name (with NO WHITESPACE!), data_file,
# surface_rep)
# and represent additional attributes (beginning with key =
# 1)
# NOTE: attribute_keys must be unique!!

# <<soil attribute 1>> , layer 1 contains soil cohesion value
attribute_key             1
attribute_name             cohesion_of_soil
data_file                  coh2.data
surface_rep                uniform_nonrational_bilinear_Bspline

# layer 2 contains another soil property: angle of internal fric-
# tion
```



```

attribute_key      2
attribute_name     angle_of_internal_friction
data_file          phi2.data
surface_rep        uniform_nonrational_bilinear_Bspline

#layer 3 contains soil density valuse
attribute_key      3
attribute_name     unit_weight_of_soil
data_file          gama2.data
surface_rep        uniform_nonrational_bilinear_Bspline

```

6.3.3 Software Design

The soil slippage model is implemented in the SoilClass. The Soil DTR is implemented in a two tier hierarchy in which the soilClient provides an interface to the underlying soil model, or Soil-Class.

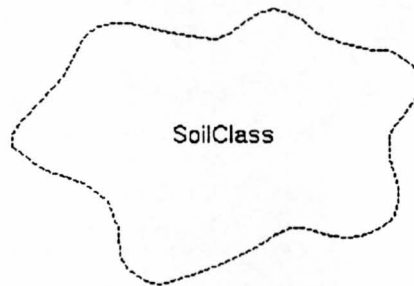


Figure 36. Booch diagram of soil model

6.3.4 Programmer's Guide

There are five overloaded *slumpSoil* methods. Three of these methods are for use with homogeneous soils in which the soil parameters cohesion, density, and internal angle of friction are constant.¹ Two of these methods are for use with heterogeneous soils and were developed more recently to accomodate non-uniform soil parameters. The latter two methods also accomodate homogenous soil as well, since the soil parameters for each terrain cell will simply be the same. The client program should call the constructor corresponding to the type of soil, homogeneous or heterogeneous, that it will be using. The class specification follows.

```

#define HOMO_VERSION          0
#define HETERO_VERSION        1

class SoilClass
{
    int      version_id ;           // either HOMO_VERSION
    or HETERO_VERSION

```

-
1. The original soil model is based on homogeneous soil. Three methods correspond to this model which uses a single value fo each soil parameter in any given terrain patch within the database. However, methods to accomodate non-uniform soil parameters were later developed .

```

        int      soil_maxx, soil_maxy; //soil patch array limits
        float    soil_dx, soil_dy;    //soil patch resolutions
        float     coh;                 // cohesion of soil
        float     phi;                 // angle of internal
friction
        float     gama;                // unit weight of soil
        float     tanF;                // tangent of friction
angle
        float     G;                  // gama*tanf.
        float     soil[MAX_DATASIZE+1][MAX_DATASIZE+1][3];
//elevation posts
        float     dt, damp;

        float     vx[MAX_DATASIZE+1][MAX_DATASIZE+1],
                vy[MAX_DATASIZE+1][MAX_DATASIZE+1],
                vd[MAX_DATASIZE+1][MAX_DATASIZE+1];
//force
        float     fx[MAX_DATASIZE+1][MAX_DATASIZE+1],
                fy[MAX_DATASIZE+1][MAX_DATASIZE+1],
                fd[MAX_DATASIZE+1][MAX_DATASIZE+1];
//cohesion parameters
        float     soil_coh[MAX_DATASIZE+1][MAX_DATASIZE+1],
                soil_tanF[MAX_DATASIZE+1][MAX_DATASIZE+1], //cor-
responds to phi
                soil_gama[MAX_DATASIZE+1][MAX_DATASIZE+1];
//thresholds
        float     new_SX[MAX_DATASIZE+1][MAX_DATASIZE+1],
                new_SY[MAX_DATASIZE+1][MAX_DATASIZE+1],
                new_SD[MAX_DATASIZE+1][MAX_DATASIZE+1];
        float     SX, SY, SD;
        float     DD, DX2, DY2, DD2;
        float     minx,maxx,miny,maxy,minz,maxz;
        int     debug_mode;            // turns on printing
when set

        int     isStable();
        float    safetyFactor(double a, double h, double dx);
        float    safetyFactor(double a, double h, double dx, int i,
int j);
        double   totalVolume();
        int      notStable(float dz, float dx);
        int      notStable(float dz, float dx,int i, int j);
        float    criticalAngle(float dz, float dx);
        float    criticalAngle(float dz, float dx,int i, int j);
        float    force(float b0, float dz, float dx);
        float    force(float b0, float dz, float dx,int i, int j);
        void     initForce();
        void     plane(double *A, double *B, double *D, int i, int j,
char m);
        double   volume(int i, int j);
        float    threshold(float b);
        float    threshold(float b, int i, int j);
        void     initMinMax();
        void     changeBoundingBox(float aPoint[3]);

```

```

        void    printBoundingBox();
        void    euler();
        void    eulerNonuniform() ;

public:
    SoilClass(int debug);
    SoilClass(int maxx, int maxy, int debug);
    SoilClass(int debug, int version); // For heterogeneous
soil, use this constructor with
//
version=HETERO_VERSION only !
    void    printSoil();

    // homogeneous version methods which assume constant valued
soil parameters
    // slumpSoil_1
    void slumpSoil(int number_of_iterations,
                    int num_in_x,                // maps to s dimen-
sion of dtdb query
                    int num_in_y,                // maps to t dimen-
sion of dtdb query
                    float x[],
                    float y[],
                    float z[]);
    // slumpSoil_2
    void slumpSoil(int number_of_iterations,
                    int num_in_x,                // maps to s dimen-
sion of dtdb query
                    int num_in_y,                // maps to t dimen-
sion of dtdb query
                    float z[]);
    // slumpSoil_3
    void slumpSoil(int number_of_iterations,
                    int num_in_x,                // maps to s dimension
of dtdb query
                    int num_in_y,                // maps to t dimension
of dtdb query
                    float points[][3]);          // interlaced array
method

    // New version methods for heterogeneous soil attributes
where
    // each value in the array s_coh[], s_phi[] or s_gama[]
may contain a
    // different value.

    // Use the constructor SoilClass(int debug, HETERO_VERSION)
with this method
    // slumpSoil_4
    void slumpSoil(int number_of_iterations,
                    int num_in_x,
                    int num_in_y,
                    float z[],                // elevation posts
only

```



```

parameter      float s_coh[],          // soil cohension
friction        float s_phi[],          // internal angle of
soil patch      float s_gama[],         // the density for the
heterogeneous version  int nonuniform) ; // 0: simplified het-
heterogeneous version                                // 1: standard

// Use the constructor SoilClass(int debug,
HETERO_VERSION) with this method
// slumpSoil_5
void slumpSoil(int number_of_itations,
               int num_in_x,
               int num_in_y,
               float points[][3] ,      //interlaced array
               float s_coh[],
               float s_phi[],
               float s_gama[],
               int nonuniform) ;
};

```

Details concerning use of each of the public methods follow.

6.3.4.1 SoilClass(int debug)

Constructor for use with the three homogeneous methods of *slumpSoil*.

6.3.4.2 SoilClass(int maxx, int maxy, int debug)

Constructor for use with the first three homogeneous methods denoted *slumpSoil_1*, *slumpSoil_2*, and *slumpSoil_3* in the class specification.

6.3.4.3 SoilClass(int debug, int version)

Constructor for use with the last two heterogeneous methods denoted *slumpSoil_4*, *slumpSoil_5* in the class specification.

6.3.4.4 void printSoil()

This method prints the elevation, cohesion, density, and internal angle of friction value contents for the current terrain patch.

6.3.4.5 void slumpSoil(int number_of_itations, int num_in_x, int num_in_y, float x[], float y[], float z[])

This method is used with homogeneous soil and is supplied the x, y and elevation components of the terrain patch. The soil parameters are obtained from the query of DTDB which includes the

layers of soil properties. The number of iterations designates the number of Euler loops in solving the differential equation for the soil model. More loops give a more precise result.

6.3.4.6 void slumpSoil(int number_of_iterations, int num_in_x, int num_in_y, float z[])

This method is used with homogeneous soil and is the same as the previous methods except that it is supplied the elevation component only.

6.3.4.7 void slumpSoil(int number_of_iterations, int num_in_x, int num_in_y, float points[][3])

This method is used with homogeneous soil and is the same as the previous two methods except that it is supplied an interlaced array of x, y and elevation components.

6.3.4.8 void slumpSoil(int number_of_iterations, int num_in_x, int num_in_y, float z[] , float s_coh[], float s_phi[], float s_gama[], int nonuniform)

This method is used with heterogeneous soil and is supplied elevation, cohesion, internal angle of friction, and density components of the terrain patch. The SoilClass(int debug, int version) constructor must be used with this method. If nonuniform is set to 0, an average value over all patch values is taken for each of the soil parameters. This average value is used to calculate the terrain patch's critical angle, slump threshold and other soil properties related factors. This averaging method is similar to the homogeneous methods except that each terrain patch has an average value instead of using a single value for the entire database. If nonuniform is set to 1, soil parameter values for individual terrain patch cells are used to calculate critical angle, slump threshold and other soil properties related factors for every cell in the patch in order to get the fine-tuned slumping effect.

6.3.4.9 void slumpSoil(int number_of_iterations, int num_in_x, int num_in_y, float points[][3] , float s_coh[], float s_phi[], float s_gama[], int nonuniform)

This method is used with heterogeneous soil and the same as the previous method except that it is supplied an interlaced array of x, y, and elevation components.

5.

7.0 Fluid DTR

7.1 Introduction to Fluid DTR

7.1.1 Background

Distributed Interactive Simulation (DIS) denotes both a broad field of simulation research and technology, and a specific architectural approach, represented by the DIS communications Protocol. This protocol and architecture are described by a U. S. military standard [IST 94]. Here we use the acronym DIS to denote any simulation to be conducted by distributed computation and whose outputs must respond to changed inputs with the same timeliness the modeled system would exhibit.

Simulating physically realistic complex fluid behaviors is one of the most challenging problems for computer graphics researchers. Such behaviors include the effects of driving boats through water, stirring liquids, blending differently colored fluids, mixing insolubles such as oil and water, rain falling and flowing on a terrain, and fluids interacting in a distributed interactive simulation. Such capabilities are useful in computer art, advertising, education, entertainment, analytical studies and training.

A physically-based model of an object is a mathematical representation of the behavior of the object based on an analytical method in physics; therefore, providing physical realism of the object's behaviors. A physically-based approach to achieve the capability of generating realistic fluid behaviors would provide a common framework and uniform treatment of such simulations, in addition to the benefits that follow from making the choice of input parameters physically relevant, physically inspired, and physically intuitive. An additional need that persists is for the capability of providing in real-time at least some, if not all, of the effects that are visually relevant to the simulation. Here we use real-time to mean that the frame rate of the physically-based modeling and simulation is at an interactive-rate of human perception. Success in this quest has eluded the computer graphics research community thus far.

Modeling and animation of fluids have captured the attention of many graphics researchers [Fournier 86, Goss 90, Kass 90, Miller 89, Peachey 86, Stam 93, Terzopoulos 89, Tonnesen 92, Ts'o 87, Wejchert 91]. However, general fluid models that are both physically realistic and computationally efficient, for real-time animation, have not been achieved. Fournier and Reeves [Fournier 86], Peachey [Peachey 86], and Ts'o and Barsky [Ts'o 87] presented alternative models based on ocean wave equations. Their approaches are limited to ocean waves. Miller and Pearce [Miller 89] presented a particle system for animating viscous fluids. Their approach represents particles throughout the volume of the fluid, incurring significant computational costs. Terzopoulos, Platt and Fleischer [Terzopoulos 89] used molecular dynamics to model the process of solids transforming into liquids. Their approach is also computationally expensive for use in a real-time, interactive simulation. Kass and Miller [Kass 90] presented a method for animating fluid using a simplification of shallow water equations. Their method is not general enough to allow for the modeling of complex phenomena that entail variations in the Reynolds number (which controls

whether a fluid is laminar or turbulent) or include moving (self-propelled) objects. Goss [Goss 90] used a particle system to model ship wakes in real-time, but his approach was not based on physics and it does not host any other fluid properties or behaviors than artificial ship wakes. Wejchert and Haumann [Wejchert 91] presented a model for inviscid irrotational flow which is only applicable to limited situations of objects in a wind field. Stam and Fiume presented a method which models turbulent wind fields [Stam 93] but lacks any physical basis.

To provide a physical foundation for general fluid animation, one must employ the Navier-Stokes Equations, which are the embodiment of Newton's second law in fluids, and the governing equations of general fluid flow. Several researchers in computer graphics have acknowledged this [Kass 90, Wejchert 91, Stam 93]. However, none of the previous models solved these equations due to the effort involved in deriving a solution method and the time needed to obtain a solution on commonly available workstations. While researchers in the discipline of Computational Fluid Dynamics (CFD) extensively study computational models of physical fluids, their goals are constrained to obtaining highly accurate and completely descriptive simulations of fluid behaviors. In contrast, the goals for many fluid applications in graphics simulation are to produce realistic looking fluid behaviors and fast calculations for real-time animation. It suffices to say that CFD researchers have not undertaken studies to develop fluid modeling tools, based on the Navier-Stokes equations, that might sacrifice completeness of description in exchange for the real-time performance, that is satisfactory for computer graphics.

This work develops a physically-based model for real-time simulation of fluids in a networked virtual environment. This section describes an approach to a general fluid model which was later implemented as a Dynamic Terrain Resource (DTR). The software implementation of this approach is referred to as the Fluid DTR.

7.1.2 Problem statement

To simulate fluid flow in a dynamic virtual environment such as a Dynamic Terrain, we should account for the changes of the fluid volume and boundaries when the fluid flows and accumulates. The fluid should be able to be generated and to flow freely anywhere on a dynamic terrain surface. A Fluid DTR should be capable of managing fluid behaviors in a distributed simulation. This is accomplished with little difficulty under the realm of the Fluid Service ("Introduction to Fluid Service" on page 66) which provides network communication and mechanisms to transmit information across the network.

7.1.3 Solution

Here, we introduce a computational fluid dynamics method using the Navier-Stokes equations. Instead of calculating the fluid behavior through a volume, that is, calculating the 3D Navier-Stokes equations, we compute the full incompressible 2D Navier-Stokes equations [Peyret 85, Wendt 92]. Then we raise the surface of the fluid according to the corresponding pressures in the flow field, thus obtaining the third dimension of the surface points. Using the pressures to simulate the fluid surface fluctuations is justified by the fact that higher pressures at the base of a fluid cause taller columns of surface above. This is due to the incompressibility of the liquid fluid. This

method reduces the time expense from the cube of the resolution to the square of the resolution without losing the 3D effects and the power of the Navier-Stokes equations. This approach achieves a real-time simulation which accommodates floating and drifting objects within the fluid.

For simulating fluid in a networked environment, we introduce several list structures for the networking of fluid. These data structure allows fluid to be easily packed into a compact data block and updated over the network.

7.1.4 Constraints/assumptions about problem

There are limitations to our fluid model. Specifically the 2D solution with the addition of pressures is not physically equivalent to the solution of 3D Navier-Stokes equations, which precisely describes the fluid behavior. As a consequence, this technique cannot be used for problems that require exact results, such as those arising in engineering applications. Also, the simulation may not be stable after a long period of time if the Reynolds number is kept too high. Finally, the fluid model does not account for phenomena such as fluid jump and fluid refraction.

7.2 Analysis

7.2.1 Real-time Fluid Model

CFD has permitted advanced simulations of flow phenomena on the computer for wide and varied applications. The areas range from aircraft and missile design to large-scale simulations of the atmosphere and ocean. Until recently, numerical methods for solving fluid-flow problems have been dominated by finite-difference approximations. These methods are powerful and play a major role in problem solutions.

There are many fluid phenomena which can not be simulated by CFD. CFD can not reproduce physics in cases where the problem itself has not been well defined. The most important example is turbulence. Most CFD solutions of turbulent flows now contain turbulence models which are just approximations of the true physics, and which depend on empirical data for various constants used in the turbulence models. Thus, all CFD solutions of turbulent flows are subject to inaccuracy, even though some models for some situations are quite reasonable [Wendt 92, page 13].

7.2.1.1 A Finite-Difference Solution of the Navier-Stokes Equations

Here, the most generally and widely used finite-difference method for the Navier-Stokes equations is emphasized. The following equations are called the Navier-Stokes equations for an incompressible flow

$$\rho \frac{Du}{Dt} = -\frac{\partial p}{\partial x} + \mu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) + \rho g_x \quad (26)$$

$$\rho \frac{Dv}{Dt} = -\frac{\partial p}{\partial y} + \mu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right) + \rho g_y \quad (27)$$

$$\rho \frac{Dw}{Dt} = -\frac{\partial p}{\partial z} + \mu \left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right) + \rho g_z \quad (28)$$

where $\frac{D}{Dt} = u \frac{\partial}{\partial x} + v \frac{\partial}{\partial y} + w \frac{\partial}{\partial z} + \frac{\partial}{\partial t}$, $u = \frac{dx}{dt}$, $v = \frac{dy}{dt}$, $w = \frac{dz}{dt}$, ρ is the density, p is the pressure, μ is the viscosity, and g_x , g_y , g_z are the gravity vectors.

The differential continuity equation for incompressible flow

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \quad (29)$$

is used together with Navier-Stokes equations to determine the relationships between velocities and pressures.

We can put the Navier-Stokes equations (1, 2, 3) and the differential continuity equation (4) into the following compact vectorial form:

$$\rho \frac{DV}{Dt} = -\nabla p + \mu \nabla^2 V + \rho g \quad (30)$$

$$\nabla \cdot V = 0 \quad (31)$$

where the gradient vector operator $\nabla = \frac{\partial}{\partial x} \mathbf{i} + \frac{\partial}{\partial y} \mathbf{j} + \frac{\partial}{\partial z} \mathbf{k}$, $V = u\mathbf{i} + v\mathbf{j} + w\mathbf{k}$, and $g = g_x\mathbf{i} + g_y\mathbf{j} + g_z\mathbf{k}$.

The Navier-Stokes equations without external forces can be written in the dimensionless form

$$\frac{DV}{Dt} = -\nabla p + \frac{1}{Re} \nabla^2 V \quad (32)$$

where Re is the *Reynolds number*. Reynolds number is a parameter that indicates mainly the viscosity of the fluid. It is defined as follows:

$$Re = \frac{VL}{\nu} \quad (33)$$

where L and V are a characteristic length and velocity, respectively, and ν is the kinematic viscosity, i.e., $\nu = \frac{\mu}{\rho}$ where μ is the viscosity of the fluid. For example, given fluid flow inside a pipe,

then L can be the diameter of the pipe and V the velocity of the fluid flow inside the pipe. ρ , L and V are considered constant in most applications, thus Re is inversely proportional to the viscosity of the fluid. So if the Reynolds number is relatively small, the fluid is viscous and the flow regime is laminar; if it is large, the fluid tends to be inviscid and the flow regime is turbulent. Therefore the results of the Navier-Stokes equations with different Reynolds numbers correspond to the behaviors of different kinds of fluids.

This approach only presents the discretization of the 2D Navier-Stokes equations to simplify the explanation. There are several approaches in computational fluid dynamics to solve the Navier-Stokes equations. Here a finite-difference solution that uses the penalization method is employed. It makes use of following equation instead of the divergence equation, i.e., differential continuity

equation.

$$\varepsilon p + \nabla \cdot \mathbf{V} = 0 \quad (34)$$

where $\varepsilon > 0$, $\varepsilon \rightarrow 0$ is a very small number introduced. Rewriting the Navier-Stokes equations explicitly as follows:

$$\frac{\partial \mathbf{V}}{\partial t} = -u \frac{\partial \mathbf{V}}{\partial x} - v \frac{\partial \mathbf{V}}{\partial y} - w \frac{\partial \mathbf{V}}{\partial z} - \nabla p + \frac{1}{Re} \nabla^2 \mathbf{V} \quad (35)$$

The error involved is in $O(\Delta x^2)$.

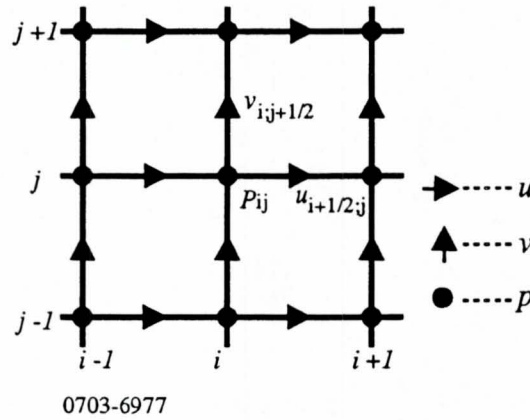


Figure 37. The staggered marker-and-cell mesh

The spatial discretization in 2D makes use of the staggered marker-and-cell mesh as shown above. Consider an explicit discretization as follows:

$$u_{i+1/2;j}^{n+1} = u_{i+1/2;j}^n + \left(-a_{i+1/2;j}^n - \Delta_x^1 p_{i+1/2;j}^n + \frac{1}{Re} \nabla_h^2 u_{i+1/2;j}^n \right) \Delta t \quad (36)$$

$$v_{i;j+1/2}^{n+1} = v_{i;j+1/2}^n + \left(-b_{i;j+1/2}^n - \Delta_y^1 p_{i;j+1/2}^n + \frac{1}{Re} \nabla_h^2 v_{i;j+1/2}^n \right) \Delta t \quad (37)$$

$$p_{i;j}^{n+1} = -\frac{\Delta_x^1 u_{i;j}^{n+1} + \Delta_y^1 v_{i;j}^{n+1}}{\varepsilon} \quad (38)$$

where i, j are fluid flow field mesh coordinates, n represents the current state and $n+1$ represents the next state after a time-step of Δt .

The difference operators Δ_x^1 , Δ_y^1 , and ∇_h^2 are defined by

$$\Delta_x^1 f_{l;m} = \frac{1}{\Delta x} (f_{l+1/2;m} - f_{l-1/2;m}) \quad (39)$$

$$\Delta_y^1 f_{l;m} = \frac{1}{\Delta y} (f_{l;m+1/2} - f_{l;m-1/2}) \quad (40)$$

$$\nabla_h^2 f_{l;m} = \Delta_{xx} f_{l;m} + \Delta_{yy} f_{l;m} \quad (41)$$

$$\Delta_{xx} f_{l;m} = \frac{f_{l+1;m} - 2f_{l;m} + f_{l-1;m}}{\Delta x^2} \quad (42)$$

$$\Delta_{yy} f_{l;m} = \frac{f_{l;m+1} - 2f_{l;m} + f_{l;m-1}}{\Delta y^2} \quad (43)$$

The terms $a_{i+1/2;j}^n$ and $b_{i;j+1/2}^n$ are the approximations of $u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + w \frac{\partial u}{\partial z}$ and $u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} + w \frac{\partial v}{\partial z}$ as defined in Equation 35.

$$a_{i+1/2;j}^n = u_{i+1/2;j}^n \Delta_x^0 u_{i+1/2;j}^n + v_{i+1/2;j}^n \Delta_y^0 u_{i+1/2;j}^n \quad (44)$$

$$b_{i;j+1/2}^n = U_{i;j+1/2}^n \Delta_x^0 v_{i;j+1/2}^n + v_{i;j+1/2}^n \Delta_y^0 v_{i;j+1/2}^n \quad (45)$$

where

$$U_{i;j+1/2} = \frac{1}{4} (u_{i+1/2;j} + u_{i+1/2;j+1} + u_{i-1/2;j+1} + u_{i-1/2;j}) \quad (46)$$

$$V_{i+1/2;j} = \frac{1}{4} (v_{i+1/2;j+1/2} + v_{i;j+1/2} + v_{i;j-1/2} + v_{i+1/2;j-1/2}) \quad (47)$$

$$\Delta_x^0 f_{l;m} = \frac{1}{2\Delta x} (f_{l+1;m} - f_{l-1;m}) \quad (48)$$

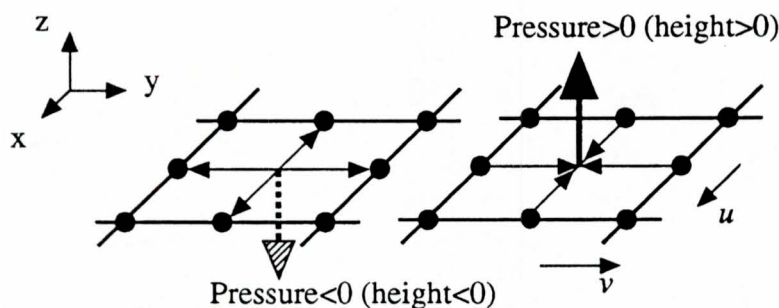
$$\Delta_y^0 f_{l;m} = \frac{1}{2\Delta y} (f_{l;m+1} - f_{l;m-1}) \quad (49)$$

All these approximations are of second-order accuracy. That is, the error involved is on the order of Δx^2 or Δy^2 . The algorithm to compute the solution to the Navier-Stokes equations then is as follows. For the known current state of the velocity vectors and pressures of the fluid flow field $\{u_{i+1/2;j}^n, v_{i;j+1/2}^n \text{ and } p_{i;j}^n\}$, the next state $\{u_{i+1/2;j}^{n+1}, v_{i;j+1/2}^{n+1} \text{ and } p_{i;j}^{n+1}\}$, after Δt time, is calculated by Equation 36, Equation 37 and Equation 38.

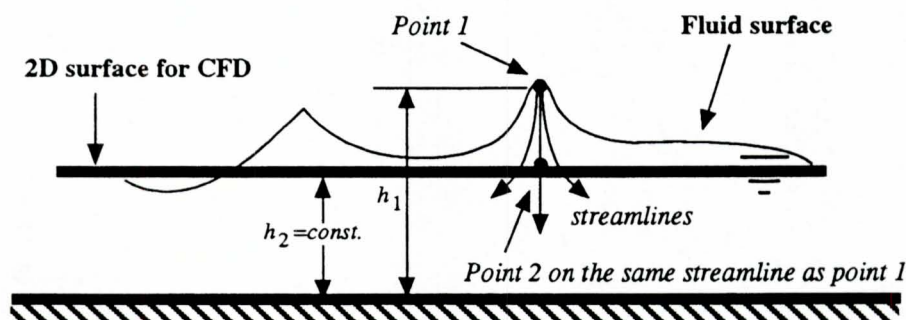
The discretization method used above is simplest for the unsteady Navier-Stokes equations. Here unsteady means the flow varies with time, i.e., the term $\frac{\partial V}{\partial t}$ is included in the equations which avoids solving the Poisson equation [Pey 85]¹.

7.2.1.2 The Third Dimension -- Pressures as Heights

Instead of calculating the fluid behavior through a volume, that is, calculating the 3D Navier-Stokes equations, we compute the full incompressible 2D Navier-Stokes equations. Then we raise the surface of the fluid according to the corresponding pressures in the flow field, thus obtaining the third dimension of surface points. Using pressures to simulate the fluid surface height can be justified by the fact that the higher pressures at the base of the fluid cause taller columns of the surface above. This is due to the incompressibility of the liquid fluid. As shown in Figure 38, "Relationships between pressure and height," a below, when fluid from neighboring points flows into one place, the pressure as well as the height of the fluid surface at that point will rise, while when the fluid at one point flows to its neighboring points, the pressure as well as the height of the fluid surface at that point will drop. This can also be explained by the Bernoulli equation, which is a basic fluid mechanics equation satisfied along two points on the same streamline [Potter 91, Fox 92] with some approximations:



a. Higher pressures cause taller columns



b. Heights and pressures from Bernoulli equation

0703-6978

Figure 38. Relationships between pressure and height

1. The most common methods used to solve the unsteady equations employ the Poisson equation for the pressures and the momentum equations for the computation of velocities.

$$\frac{V_1^2}{2} + gh_1 + \frac{p_1}{\rho} = \frac{V_2^2}{2} + gh_2 + \frac{p_2}{\rho} \quad (50)$$

where V_1 and V_2 are the velocities at point 1 and point 2 respectively, g is the gravity, h_1 and h_2 are the heights, p_1 and p_2 are the pressures, and ρ is the density of the fluid (Figure 38, "Relationships between pressure and height." b). To simplify the analysis, we consider the velocities at the two points to be equal, i.e., $V_1 = V_2$, and we assume point 1 is on the surface of the fluid and point 2 is on the 2D surface we used to calculate the 2D Navier-Stokes equations, as shown in Figure 39, "Fluid Source List and Tip List structure." b. As we can see, the surface of the fluid has no pressure (i.e., $p_1 = 0$) and the height of the 2D surface is constant ($h_2 = \text{constant}$). Therefore, for our purpose, the Bernoulli equation can be simplified as follows:

$$h_1 = \frac{p_2}{g\rho} + h_2 \quad (51)$$

Since g , ρ and h_2 are all constants, the height h_1 is proportional to the pressure p_2 . In particular, when $p_2 = 0$, $h_1 = h_2$. Empirically, the demonstrations of this physically realistic real-time simulation is further justification. This reduces the expense from the cube of the resolution to the square of the resolution without losing the three dimensional fluid surface behaviors and the power of the Navier-Stokes equations. The main idea is that pressures can be used to simulate the heights of the fluid surface points, and thus the computation is significantly reduced so that the general Navier-Stokes equations can be applied to model and simulate fluid surface behaviors in real-time for computer graphics. The applications are wide-ranging due to the generality of the Navier-Stokes equations.

In our approach, we use the corresponding pressures in a 2D fluid flow field to simulate the fluid flow in 3D. Therefore, we only need to compute the 2D Navier-Stokes equations. We have provided a finite-difference solution of the Navier-Stokes equations. In order to use integer indices, we multiply every index by two to avoid the $1/2$ used in the indices in the theory.

After we calculate each state of the velocity vectors and pressures of the 2D fluid flow field, we can render the current frame of the velocity field. For a given grid (i, j) in the flow field,

$$u_{i,j} = \frac{u_{i+1/2,j} + u_{i-1/2,j}}{2} \quad (52)$$

$$v_{i,j} = \frac{v_{i,j+1/2} + v_{i,j-1/2}}{2} \quad (53)$$

we can draw a velocity vector from (i, j) to $(i+u_{i,j}, j+v_{i,j})$. By elevating the grid (i, j) in the third dimension to some scaled value of $p_{i,j}$, we can draw the fluid velocity field in 3D. Therefore as the real-time calculations and rendering progress, we can animate the velocity vectors of the fluid

flow field. By shading and lighting the surface of the flow field, we can obtain a frame of the simulation of the channel flow.

So for a fluid flow, we have obtained not only the simulation of the fluid surface, but also the velocity field on the surface, and thus providing the velocities of all points on the fluid. This information is very important and can be applied to simulate objects floating with the speed of the fluid.

7.2.2 Fluid Model in a DT Simulation

The pressure-height 3D surface behavior does not include the fluid volume and boundary changes of the fluid flow, making it difficult to simulate flows that depend on the simulated environment. In many applications, such as simulations involving a dynamic terrain, fluid flow interacts with the environment. For example, in [Li 93] a bulldozer may breach a dam resulting in water flowing over soil which is being continually modified. Therefore, fluid conservation and the interaction of the flowing fluid with its environment are important issues to be addressed. We now introduce a method which calculates the fluid flow and volume conservation necessary for complementing the physical fluid surface behavior simulation. Hence, our general fluid model allows fluid to be generated anywhere on a terrain surface and the flow follows the 3D contours of the environment. It allows changing boundaries and accounts for depth of the fluid. Data structures were designed for networking of fluids in a distributed simulation. With these capabilities, the fluid model can be integrated into the DT testbed within a DIS.

7.2.2.1 Fluid Flow in DT

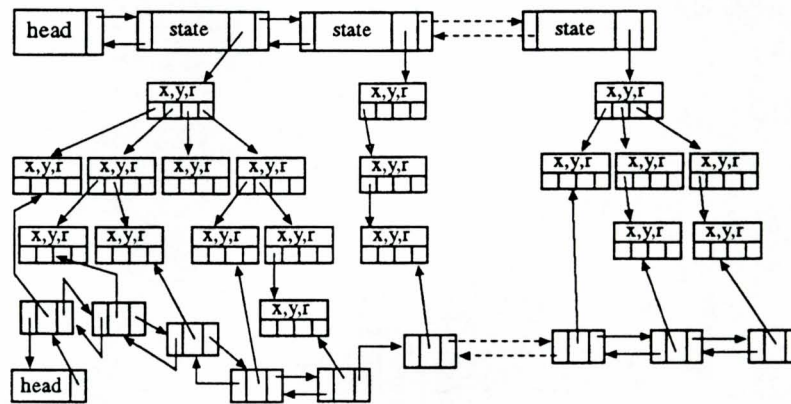
There are three main structures in this generalized fluid model for displaying and networking: the Source List, the Tip List, and the End List. The Source List manages the fluid sources, the Tip List manages the fluid flow from sources to the destinations where the flow accumulates, and the End List manages the fluid accumulation. These three main structures are connected by fluid flow points on the terrain. The terrain is represented by elevation posts and each elevation post corresponds to the location of a point. The connected fluid flow points form a path of fluid flow. Following a fluid source in the Source List, we can step through the fluid flow points to the locations in the Tip List where the fluid is expanding or to the locations in the End List where the fluid is accumulating.

7.2.2.2 Source

The Source List contains fluid source pointers and state information. A fluid source pointer provides the location (x,y) and the rate of flow (r) of a fluid source on a terrain (Figure 39, "Fluid

Source List and Tip List structure.”). Fluids are generated from sources in the Source List.

Source List



Tip List

0703-7000

Figure 39. Fluid Source List and Tip List structure

7.2.2.3 Tip

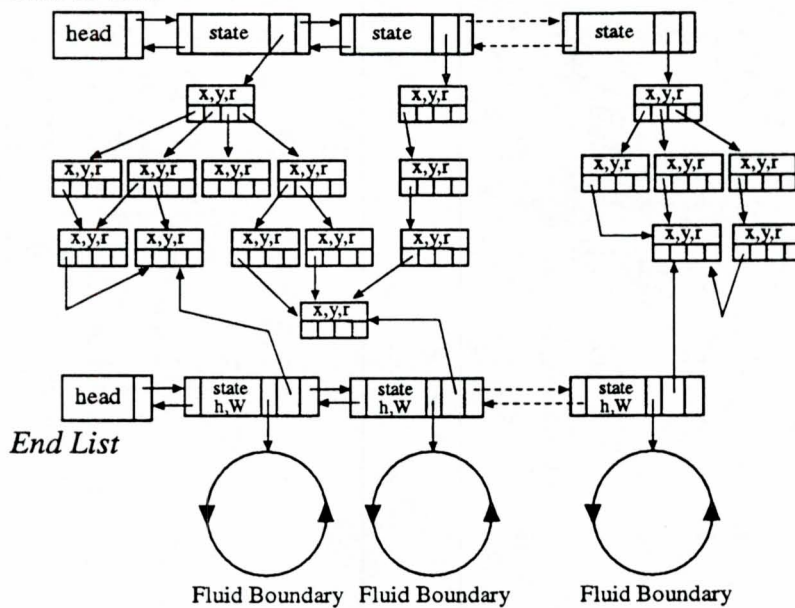
The Tip List is a temporary list connecting all points which are currently at the tips of the fluid flow (Figure 39, “Fluid Source List and Tip List structure.”). It allows us to calculate the leading edge of the fluid until it arrives at the end points where it will accumulate. Flow rates are calculated and carried along the fluid flow with the tips. Notice that those fluid points between a fluid source and a fluid tip are neither in the Source List nor in the Tip List. They are points along the path of fluid flow from the source to the tip. When a fluid tip expands to a point where the fluid ceases to flow because it is lower than all its surrounding points, we reclassify the tip as an end point where fluid accumulates. The end point is added to the End List. The End List contains locations where fluid is accumulating (Figure 40, “Fluid End List structure.”).

7.2.2.4 End

The End List contains fluid states, fluid surface heights (h), fluid surface areas (W), pointers to fluid boundaries, and pointers to fluid end points (x, y) and flow rates (r). Fluid state information tells whether the current fluid volume is growing. Fluid surface height allows us to draw the surface at its corresponding height. Flow rate and surface area together determine the fluid height changes for any particular instant of time. The fluid boundary, or perimeter, is a linked list of boundary points of the fluid surface (Figure 41, “Fluid boundary structure.”). The structure is convenient for DIS because we only need to send the fluid heights and boundaries across the network to describe the same fluid areas and depths on different simulators. Here we point out that the Source List, the Tip List and the End List are coexistent in a fluid flow and terrain environ-

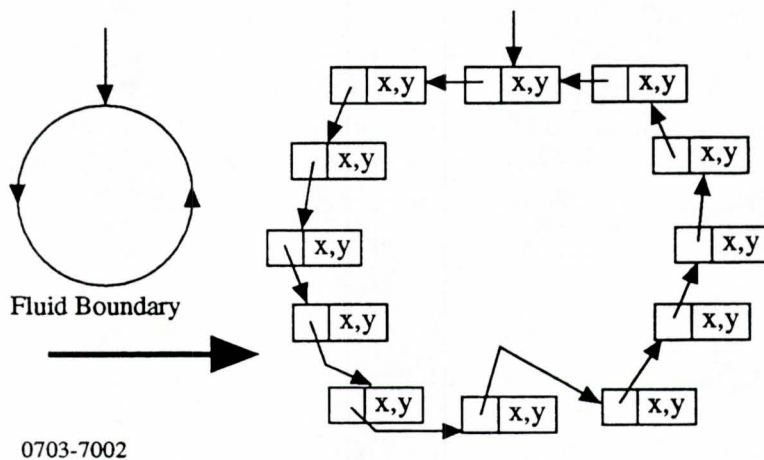
ment, although to simplify the explanation we did not draw them together.

Source List



0703-7001

Figure 40. Fluid End List structure



0703-7002

Figure 41. Fluid boundary structure

7.3 Implementation

7.3.1 Problem Statement

Next we discuss the method to simulate fluid in the networked environment. Here we used a flat surface to represent the fluid area. The flat surface fluid model is useful in low fidelity networked fluid simulations where fluids are used for the purpose of orientation. Fluid flow calculation includes adding fluid at source points (`fluidProcessInput`), calculating current fluid flow (`fluidStateFluid`), and rendering current fluid surfaces (`fluidDrawFluid`). Therefore, the general program is as follows:

```
fluidFlow()
{
    /* PROCESS: main procedure of fluid simulation. */

    /* process the input information */
    fluidProcessInput();

    /* calculate the current fluid field */
    fluidStateFluid();

    /* animate the fluid */
    fluidDrawFluid();
}
```

7.3.1.1 Process the Input Information: `fluidProcessInput()`

Once a source and a rate of flow is determined, a new fluid source node is created in the Source List and in the Tip List. Initially, a new fluid source state is set to active.

7.3.1.2 Calculate the Current Fluid Field: `fluidStateFluid()`

Calculating current fluid flow includes updating the fluid source when there is a fluid source added or removed, updating the fluid expanding at the fluid tips, and calculating the fluid accumulations at the end points of the fluid flow. The structure is as follows:

```
fluidStateFluid()
/* PROCESS: calculate next state of fluid.*/
{
    /* fluid flow and expanding at the tips */
    fluidTipProcess();

    /* fluid surfaces raise because of accumulations */
    fluidEndProcess();

    /* remove and update dry sources */
    fluidSourceProcess();
}
```

7.3.1.2.1 Update a Source: fluidSourceProcess()

When a user specifies a source and a rate of flow, a new fluid source node is created in the Source List and in the Tip List. Initially, the fluid source state is set to active. When a fluid source is stopped or removed, that is, when there is no more fluid flowing from the current fluid source, we set the state of the source to inactive (state = 0), and then iteratively trace the fluid flow from the inactive source points to the end points, removing all the fluid flow points along the path. Note that if two sources flow together at some intermediate point, our data structure maintains two separate entries, one associated with each source. Only the entry associated with the inactive source is removed in this process. At the end point of the fluid flow, only the flow rate of the traced source is deducted. Thus we can have many new user input fluid sources generating fluids and many dried fluid sources being removed.

7.3.1.2.2 Flow at the Tips: fluidTipProcess()

Fluid tips in the Tip List are processed by expanding to new locations, that is, new tips. When this occurs the current tips are delinked from the Tip List and become inner points of the fluid flow. Every fluid tip is a temporary structure which is used only once. It is used to find the next positions to which the fluid flows from its current position according to the environment (i.e., the shape of the terrain, which, in this case, consists of elevation posts). A tip searches its surrounding points (namely, elevation posts), and the next positions are those points which are lower than the current point. The fluid flow rate is divided by the number of the next positions and the result of the division is the flow rate passed on to these next positions. If this position is a local lowest point on the terrain, then there is no next position. In this case, fluid will start to accumulate and this point is added to the End List. Otherwise new fluid tips are generated as children of the current tip and added to the current Tip List, and the current fluid tip is removed from the Tip List becoming an inner point of the fluid flow.

7.3.1.2.3 Accumulate at the Ends: fluidEndProcess()

The points in the End List are points where fluid accumulates. The fluid surface heights, surface areas, surface boundaries, and the mergers of fluid boundaries are updated by processing the End List.

We have said that the fluid flow rate is carried along fluid flow points to the end points. Given the fluid flow rate r at an end point, the fluid volume (Ψ) to be added to the current location at an iteration is

$$\Psi = r \cdot \Delta t \quad (54)$$

If the elevation of the environment around the current fluid surface is very close to the height of the current fluid surface, the fluid will expand to its surrounding area at the rate corresponding to the volume Ψ . Assuming that without raising the fluid surface, each elevation post (point) will take a unit of fluid volume and the surrounding area has P points to receive a portion of Ψ , then the fluid will expand to P^* ($P^* \leq \Psi$) points. The fluid surface area (W) will be updated as $W = W + P^*$. The neighboring points of the current fluid area are randomly chosen. If ($P < \Psi$), then ($P^* =$

P), else ($P^* = \Psi \leq P$). If the surrounding points of current fluid surface are higher than the fluid surface, the fluid surface will rise according to the added volume and surface area. The surface height rise Δh is calculated by the following equation:

$$\Delta h = (\Psi - P^*) / W \quad (55)$$

When the fluid has a boundary point which is lower than the current fluid surface height by a pre-set threshold, we reclassify current boundary point as a new fluid source by adding it into the Source List and the Tip List with the current flow rate. Then the current fluid will stop expanding. To allow the current fluid to continue expanding, we can generate a new fluid source with only a portion of the current flow rate.

When two fluid surfaces start to intersect with each other, we merge these two surfaces into one fluid surface. You can see that when two surfaces start to intersect, their surface heights are very close to each other. The intersection is detected by a marker method instead of calculations of boundary intersections. Calculating intersections between fluid surfaces is very expensive. When fluid expands to a post, it marks the current post. When fluid expands to a post which is already marked, it detects an intersection, and the current post is used to find the intersected surface.

7.3.2 User's Guide

The fluidDTR is responsible for making calculations of where the fluid flows and getting the appropriate information to the Fluid Service ("Introduction to Fluid Service" on page 66) for subsequent distribution to other machines on the network. Though not strictly a class, the fluidDTR contains both the fluid model and interfaces to the shared environment. The flat surface fluid model described above is implemented within the HeuristicFluidModel class and connection to the shared environment is achieved via the FluidServiceClientInterface and the TerrainServiceClientInterface.

7.3.2.1 Configuration

Two separate configuration files are required to run the fluidDTR, one for the fluid model and one for the DTR itself. Each is detailed below.

7.3.2.1.1 Configuring the HeuristicFluidModel

A file called fluidModel.cfg is needed to setup the HeuristicFluidModel. A copy of the file is shown below with additional comments in italics.

```
HEURISTIC_FLUID_MODEL_CONFIG_FILE
#-----
Note that this configuration file must begin with
'HEURISTIC_FLUID_MODEL_CONFIG_FILE'.

# -----
#
```

```

# extent quadrilateral, in clockwise order from s,t (0,0) corner
# NOTE: the extents must be defined before the spacing
#
This extent quadrilateral corresponds to the notion used by the
DynamicTerrainDatabase (used throughout the system). Note, how-
ever, that the fluidDTR for its own purposes chooses to think in
terms of an axis-aligned rectangle (similar to a SW and NE corner).
p1_x          0.0
p1_y          0.0
p2_x          0.0
p2_y          256.0
p3_x          256.0
p3_y          256.0
p4_x          256.0
p4_y          0.0

# -----
#
# post spacing in parametric s (p1..p4) and t (p1..p2) directions
#   in database units
#
These values are in terms of the units of the underlying terrain
database. In essence, they allow control over the 'resolution'
used by the fluid model. Given that the model itself thinks in
terms of an axis-aligned rectangle (of terrain), spacing_s can be
thought of as the 'x spacing' and spacing_t the 'y'. In addition,
since the current implementation is designed for regular spacing
both of these values should be the same. However, they are kept
separate to provide for future development.
spacing_s      1.0
spacing_t      1.0

```

7.3.2.1.2 Configuring the fluidDTR

A file called fluid.cfg is needed to setup the fluidDTR program. A copy of the file is shown below with additional comments in italics.

```

FLUID_DTR_CONFIG_FILE
#-----
Note that this configuration file must begin with
'HEURISTIC_FLUID_MODEL_CONFIG_FILE'.

# -----
#
# desired frame rate for DTR
#
This is the desired update frequency to run the model.
Hz 2.0
# -----
#
# number of cycles of fluid model between updates to
#   fluid service
#

```

This is the number of cycles of the fluid model between updates to the FluidService. Generally, it is not necessary to output the fluid to the service more than about once per second or so, but it can be configured to suit the needs of the scenario. For example, if the model is run 4 times per second (Hz=4 above), and the num_cycles_per_update is 2, then new fluid state will be output to the FluidService approximately every half second.

num_cycles_per_update2

```
# -----
#
# max number of samples (in each direction) per
# terrain service query
#
This value is used to control the maximum size of a query to the
terrain service for elevation. Note that this is in each direction.
max_samples_per_query40
```

```
# -----
#
# fluid sources to be added at initialization, in the form:
#
#      fluid_source    <x>    <y>    <rate>
#
# x and y are in "grid" units, i.e., indices into the elevation
# grid used by the model (based on extents and spacing)
# 0.0 < rate <= 1.0
#
Fluid sources are added to the model based on the tuples below.
fluid_source    128    128    0.3
fluid_source    164    64    0.1
```

7.3.3 Software Design

The diagram below shows the relationships between the components of the fluidDTR.

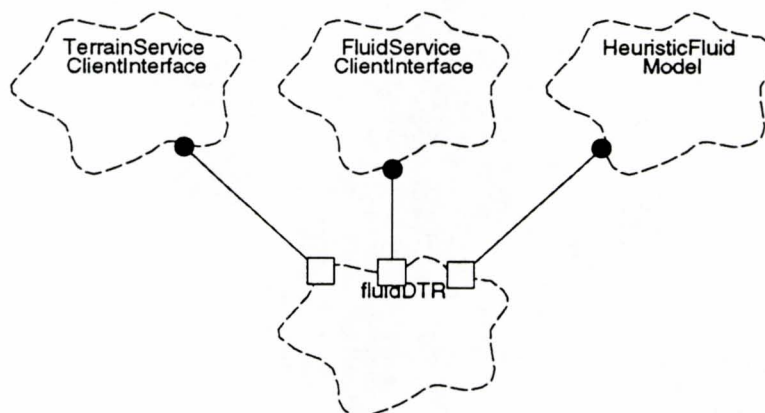


Figure 42. Components of the fluidDTR

7.3.4 Programmer's Guide

The most important thing to mention is that the software for handling and distributing fluids within DT is relatively immature. Many other components of the DT system have been used for quite some time; thus, they have stabilized. The fluids are a recent addition. Consistent with our iterative, incremental design approach this first cut has modest requirements. The most important is to flesh out the mechanisms for adding fluids to the shared environment with less concern for perfection on the first try.

The `HeuristicFluidModel` class changes the original standalone implementation of the flat surface fluid model very little. It encapsulates the fluid functionality and provides some additional methods to manage a changing elevation grid. A better implementation of this model, as well as implementations of other fluid models, are logical extensions of the work.

A constraint on the current implementation of networked fluids is that only a single fluid, water, is supported. Clearly, this should be rectified in any future implementation.

8.0 Crater DTR

8.1 Introduction to Crater DTR

8.1.1 Background

A Dynamic Terrain (DT) has the capability of rearranging the terrain surface within a real-time graphical simulation. Munition explosions that impact a terrain should result in craters. This section describes an approach to a kinematic crater model which was one of the first implementations of a Dynamic Terrain Resource (DTR). The software implementation of this approach is referred to as the Crater DTR.

8.1.2 Problem Statement

A Crater DTR should be capable of generating a crater at the location of impact of a munition or detonation of a mine on a simulated terrain. This is accomplished with little difficulty under the realm of DT Services which provides DIS Detonation PDU updates and flexible terrain queries.

8.1.3 Solution

The Crater DTR encapsulates the cratering algorithm which changes terrain elevation based on the location of munition impact and the underlying gridded terrain elevation. The algorithm simply depresses terrain beneath ground zero and raises the terrain along the sides to create a circular berm. Soil volume is not conserved during the cratering process. The Crater DTR generates the same size crater for every detonation.

The next generation Crater DTR will vary crater dimensions as a function of soil type, depth of penetration, and munition.¹ The basic Crater DTR discussed in this section will remain intact since terrain patch determination and detonation tracking are still necessary. The terrain modification method of the current Crater DTR will be substituted with a higher fidelity model.

It is possible for the Crater DTR to simultaneously change multiple attributes such as elevation and thermal values. The Crater DTR has an option to create a generic thermal footprint as a result of a munition explosion. There is no analytical basis for the thermal model implemented in the current Crater DTR; however, this model can easily be replaced when an appropriate algorithm is found.

The Crater DTR exhibits the planned flexibility of a typical DTR. First implementations are approximations at best. However, they can be substituted with more accurate physically-based models as these models come into existence. The Crater DTR's modification to the terrain's ele-

1. Current work focuses on an upgrade to the cratering algorithm and involves collaboration with U.S. Army's Waterways Experiment Station.

vation attribute will see a more accurate model with implementation of the WES cratering algorithm. Following this iteration could be a physically-based cratering model. The same growth is possible for the terrain's thermal attribute modification by a detonation.

8.1.4 Constraints/Assumptions

The Crater DTR implementation operates under the following constraints and assumptions:

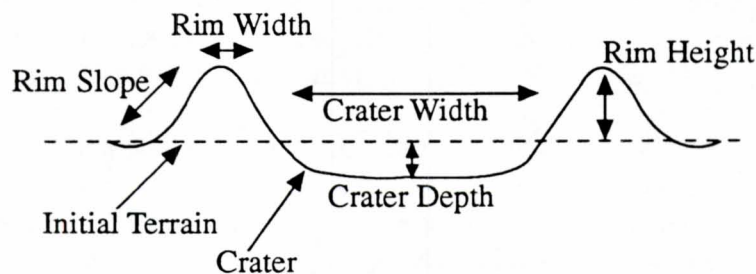
- For DIS applications, the Crater DTR is a client to both Terrain and Entity Service. This implementation is called the Crater DTR. Terrain Service provides the most recent terrain data and incorporates any new changes made by the Crater DTR into the global terrain. Entity Service provides the model with updates on detonations.
- The Crater DTR uses a two-tier implementation that separates the actual model, Crater DTR, from the Crater DTR interface to DT Services. This design offers the flexibility of alternate implementations. It is not necessary for the Crater DTR to be a client to DT's Services.

8.2 Analysis

The following analysis pertains to a Crater DTR that modifies the elevation attribute of a terrain. Clients to Entity Service receive updates on all simulation entities. The Crater DTR ignores all except detonation PDUs which invoke the Crater DTR. Upon receipt of a detonation PDU, a kinematic heuristic is used to create a soil displacement in the shape of a crater.

8.2.1 Soil Displacement

The crater profile is described by its width, depth, rim height, rim width, and rim slope in the figure below. These parameters are adjusted to achieve the desired appearance.



0703-6981

Figure 43. Crater Profile and Parameters

The terrain patch affected by a crater is determined from the crater profile parameters listed above. A query for the selected area is made to Terrain Service before applying the heuristic to the gridded elevation values returned by the query.

Since the radius of the crater, instead of its width, is often used in the calculations,

$$CraterRadius = \left(\frac{CraterWidth}{2} \right) \quad (56)$$

The change in the elevation attribute is computed as an offset. This offset is added to the sampled terrain data. The depth of the crater has been set to half the rim height. Therefore, as the crater profile is stepped across in direction x , the offset while within the crater width is negative:

$$Offset(x) = -\left(\frac{RimHeight}{2} \right) \quad (57)$$

for

$$x < CraterRadius \quad (58)$$

while the offset while within the rim width is positive:

$$Offset(x) = RimHeight \quad (59)$$

for

$$CraterRadius < x < (CraterRadius + RimWidth) \quad (60)$$

The rim changes in height from a maximum of $CraterHeight$ at the $RimWidth$ dropping linearly to zero at $CraterRadius/RimSlope$. The descent to zero is moderated by the $RimSlope$ where larger slope values cause the berm to descend more rapidly. Note that a $RimSlope$ value of unity creates a crater whose rim disappears at half the size of the crater's width. The $Offset$ outside the rim is expanded as:

$$Offset(x) = \left[\frac{\left(\frac{CraterRadius}{RimSlope} \right) - [x - (CraterRadius + RimWidth)]}{\frac{CraterRadius}{RimSlope}} \right] RimHeight \quad (61)$$

for cell values outside of the rim limits

$$(CraterRadius + RimWidth) < x < \left(\frac{CraterRadius}{RimSlope} \right) \quad (62)$$

where cells outside of these limits have an offset of zero and go unaffected.

8.2.2 Thermal Attribute

The terrain thermal attribute modification implements the same algorithm as the elevation attribute discussed above; however, the $RimHeight$ parameter is essentially an intensity value. Obviously this model is oversimplified, yet it's function in the DT testbed is more important. It shows that modification and update to multiple terrain attributes is possible. These modifications are visible to interested players within a simulation.

8.3 Implementation

8.3.1 Problem Statement

The cratering algorithm described above relies on DT Services to accomplish its task. As a client of Entity Service, it obtains detonation updates using the Entity Service Client Interface. To obtain a terrain patch centered about the point of impact, it is a client to Terrain Service using the Terrain Service Client Interface. This interface to Terrain Service also provides the capability of

updating a terrain patch which is then integrated into the global database by Terrain Service. Thus, the Crater DTR provides the terrain changing algorithm and is shielded from the tasks of listening to the network for incoming PDUs and interpolating gridded terrain.

8.3.2 User's Guide

Two implementations are possible for the crater model -- alone and in conjunction with DT Services. The Crater DTR provides an interface to the crater model much as the Terrain Service Client Interface provides a link to Terrain Service. The Crater DTR works in conjunction with both Entity and Terrain Service. Once Entity and Terrain Service are instantiated during the simulation, the Crater DTR is invoked which instantiates Entity and Terrain Service Client Interfaces and the CraterModel. The implementor does not have to create any code and simply edits the configuration file.

8.3.2.1 Configuration File

The Crater DTR configuration file should be edited prior to a simulation. The line consisting of 'CRATER_CONFIG_FILE' should appear as the first line in the file, otherwise the file is ignored. A sample configuration file follows:

```
CRATER_CONFIG_FILE
debug          1
do_heat        0
samplesize     1
Hz             1
```

A value of 1 for debug initiates the printing of crater statistics. The do_heat value indicates whether or not to modify thermal attributes as well as elevation attributes. The sample size controls the resolution of the Terrain Service database query. Both x and y resolution of the patch are the same. The Hz value controls the Entity Service update rate. Thus, a rate of 1 Hz results in updates for detonation PDUs no more than once a second.

8.3.3 Software Design

The crater model is implemented in a two tier hierarchy in which the Crater DTR provides an interface to the underlying model, the CraterModel.

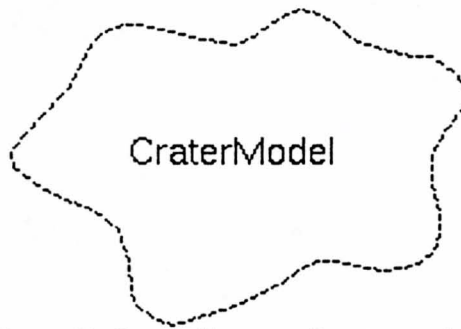


Figure 44. Booch diagram of crater model

The CraterModel, EntityServiceClientInterface, and TerrainServiceClientInterface are instantiated upon execution of the Crater DTR.

8.3.4 Programmer's Guide

The class specification for the CraterModel follows.

```
class CraterModel
{
    protected:
    float distance2D(float s, float t, float u, float v);
    void printArray(int ns, int nt, float points[]);
    void printArray(int ns, int nt, float points[][3]);

    public:
    CraterDTR();

    void findCraterParams(
        int munition_type[5],
        float* radius,
        float* rim_thickness,
        float* crater_height,
        float* side_slope);

    void findThermalCraterParams(
        int munition_type[5],
        float* radius,
        float* rim_thickness,
        float* crater_height,
        float* side_slope);

    void findQueryRecord(
        float center[3],      // location of crater center
        float radius,         // radius of crater
        float rim_thickness,  // width of flat rim on crater top
```



```

float side_slope,      // grade of side slope (see below)
int meters_per_sample, // spacing between sample control points
unsigned char *ns,     // number of points in s direction
unsigned char *nt,     // number of points in t direction
float p1[2], float p2[2], // dtdb query points
float p3[2], float p4[2]);

void generateCrater(
    float center[3],      //location of crater center
    float radius,         // radius of crater
    float rim_thickness,  // width of flat rim on crater top
    float crater_height,  // max displacement height
    float side_slope,     // grade of side slope (see below)
    unsigned char ns,     // maps to s dimension of dtdb query
    unsigned char nt,     // maps to t dimension of dtdb query
    float p1[2],          //bounding points for dtdb query
    float p2[2],
    float p4[2],
    float points[][3]);   //interlaced x, y, elevation array

void generateCrater(
    float center[3],      // point of crater center
    float radius,         // radius of crater
    float rim_thickness,  // width of flat rim on crater top
    float crater_height,  // max displacement height
    float side_slope,     // grade of side slope (see below)
    unsigned char ns,     // maps to s dimension of dtdb query
    unsigned char nt,     // maps to t dimension of dtdb query
    float p1[2],          //bounding points for dtdb query
    float p2[2],
    float p4[2],
    float z[]);           //elevation points only
};

```

Details of the public methods follow.

8.3.4.1 CraterDTR()

This is the constructor.

8.3.4.2 void findCraterParams(int munition_type[5], float* radius, float* rim_thickness, float* crater_height, float* side_slope)

Given the munition type, crater radius, rim thickness, height, and side slope are returned. In this simple model, the munition type is not applied and all munitions create a generic crater with

radius of 2, height of 1, no rim thickness (simply a peak), and a side slope of 1. The unity slope value causes the rim to disappear at half the crater's width.

8.3.4.3 void findThermalCraterParams(int munition_type[5], float* radius, float* rim_thickness, float* side_slope)

This method act much like the previous; however it creates 2 generic thermal footprints. For non-US battlefield support munitions, the thermal crater has a radius of 0.2, peak value of 20, no rim thickness, and a slope of 0.1; while US battlefield support munitions have identical parameters except for a peak value of 30.

8.3.4.4 void findQueryRecord(float center[3], float radius, float rim_thickness, float side_slope, int meters_per_sample, unsigned char *ns, unsigned char *nt, float p1[2], float p2[2], float p3[2], float p4[2])

Given the detonation's location and the crater parameters determined in the previous two methods, this method determines the terrain patch boundary points p1 through p4 and the number of samples in X and Y directions. The number of samples is influenced by the resolution spacing in the configuration file (Section 8.3.2.1 Configuration File).

8.3.4.5 void generateCrater(float center[3], float radius, float rim_thickness, float crater_height, float side_slope, unsigned char ns, unsigned char nt, float p1[2], float p2[2], float p4[2], float points[][3])

Given interlaced ("The Class Definition" on page 17) x, y and terrain patch elevation or thermal values and crater parameters, this method adjusts the patch values according to the crater model. These new values are available for integration into the global database. It is assumed the patch is square; therefore, spacing is equal in all directions of the patch and only 3 bounding points are passed to the method.

8.3.4.6 void generateCrater(float center[3], float radius, float rim_thickness, float crater_height, float side_slope, unsigned char ns, unsigned char nt, float p1[2], float p2[2], float p4[2], float z[])

Given terrain patch elevation or thermal values and crater parameters, this method adjusts the patch values according to the crater model. These new values are available for integration into the global database. It is assumed the patch is square; therefore, spacing is equal in all directions of the patch and only 3 bounding points are passed to the method.

9.0 Track DTR

9.1 Introduction To the Track DTR

9.1.1 Background

A Dynamic Terrain (DT) has the capability of rearranging the terrain surface within a real-time graphical simulation. In an effort to make a simulation appear more realistic, moving vehicles should leave track marks. This section describes an approach to a kinematic vehicle track model which was implemented as a Dynamic Terrain Resource (DTR). The software implementation of this approach is referred to as the Track DTR.

9.1.2 Problem Statement

A Track DTR should be capable of following a vehicle's movement across a simulated terrain and laying a footprint, or track, in its path. This is accomplished with little difficulty under the realm of DT Services which provides entity updates and flexible terrain queries.

9.1.3 Solution

The Track DTR encapsulates the track-making algorithm which changes terrain elevation based on the location of the track-making vehicle and the underlying gridded terrain elevation. The algorithm simply depresses terrain beneath the track and raises terrain along the sides of the depression marks. Soil volume is not conserved during the track-making process. However, a physically-based track model could replace this simple heuristic. The basic Track DTR could remain intact since terrain patch determination and vehicle bookkeeping would still be necessary.

It is possible for the Track DTR to simultaneously change attribute values such as elevation, thermal intensity, and/or soil strength. Extensions to the current model could include modification to thermal and soil strength terrain attributes with the addition of appropriate algorithms for modeling these changes. A similar process can be used to compute changes to additional attributes such as soil strength and thermal intensity. For example, soil strength values could be increased for the depression component and decreased for the berm component.

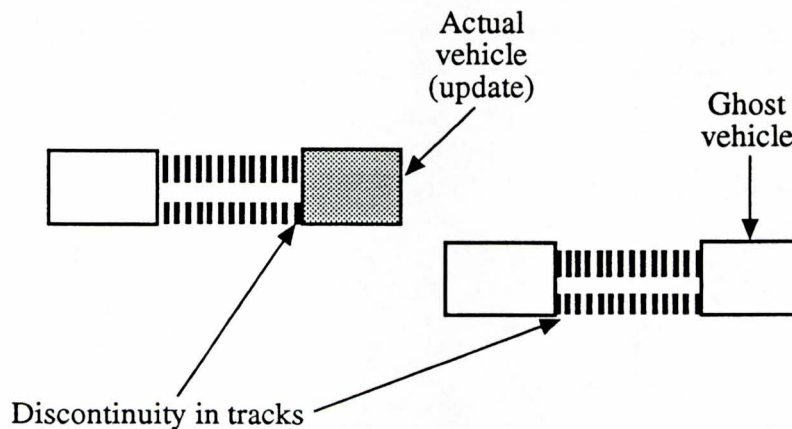
9.1.4 Constraints/Assumptions

The Track DTR implementation operates under the following constraints and assumptions:

- For DIS applications, the Track DTR is a client to both Terrain and Entity Service. This implementation is called the trackDTR. The Terrain Service provides the most recent terrain data and incorporates any new changes made by the Track DTR into the global terrain. Entity Service provides the model with updates on simulation entities. Tracks are left behind entities of interest which are specified by the implementor in a

configuration file prior to the simulation exercise.

- The Track DTR uses a two-tier implementation that separates the actual model from the interface to the DT Services. This offers the flexibility of alternate implementations. It is not necessary for the Track DTR to be a client to DT's Services. It can become part of another simulation, such as the bulldozer, to allow finer control over terrain queries.
- To function in a DIS environment, the Track DTR handles several vehicles, each creating its own tracks. This requires keeping a list of "active" vehicles. With each simulation cycle, the Track DTR adds new vehicles or drops vehicles which have not been active for a specified amount of time. Doubly-linked list and dictionary data structure tools were developed on the DT project. These were used to perform the bookkeeping operations mentioned above.
- Dead-reckoning can cause problems because it is possible for the "ghost" of a vehicle to travel along a slightly incorrect path due to use of its own local velocity.¹ When an actual entity update from the player arrives, the vehicle appears to jump to its correct location. The track behind the vehicle will show this discontinuity. The ghost vehicle's path could be smoothed before being laid by determining a trajectory based on a history of previous positions relative to the current position. This smoothing will not be performed in the Track DTR. This is accomplished with a linear or polynomial interpolation within the Entity Service.



0703-6857

Figure 45. Dead-reckoning Anomaly

- The terrain query is performed such that the same point in the query buffer is used as the center of the section of track laid by the vehicle with each cycle. However, when the vehicle turns, the track line segments will not be parallel to the direction of motion the vehicle is currently reporting. This anomaly is proportional to the turning of the

1. In a player/ghost strategy, dead-reckoned approximations of entities, or ghosts, are used to conserve network traffic. Each ghost has a concrete model, or player, from which it receives updates. The player is responsible for computing precise state information. Whenever the difference of the states between a player and ghost exceed an allowance, a message is transmitted to the ghost so it can update its state.

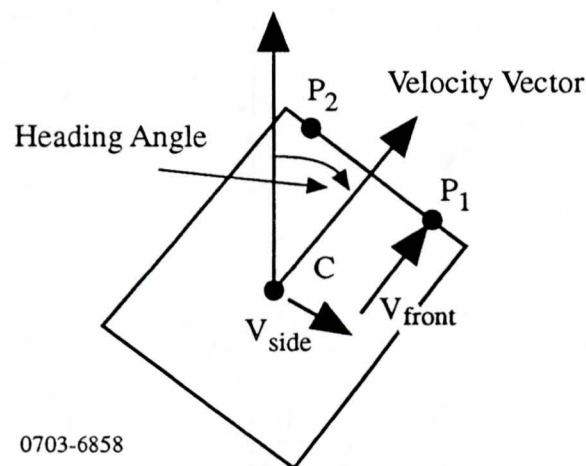
vehicle and the simulation time step.

9.2 Analysis

The following analysis considers a vehicle with two tracks, such as an M1 tank. However, this model allows for any number of tracks. A possible extension of this model is use with wheeled vehicles; however, turning wheels relative to the vehicle's orientation has not been considered. Clients of Entity Service receive updates on all simulation entities. The Track DTR ignores entity updates for those types that are not indicated in the configuration file (see Section 9.3.2.1). Obviously the specified entities must be ground-based. Matching between new and old records in its history list is accomplished with a unique entity number composed of both host and entity identification. Items lacking Entity Service updates within a set number of Track DTR cycles are removed from the list.¹ When an incoming entity update matches a record in the list, its current position is compared to its previous position. If the position changed by at least the resolution set in the configuration file, the track creation algorithm is invoked. Details of the algorithm follow.

9.2.1 Instantaneous Track Points

Once the entity's position, C , and orientation are updated, instantaneous track points, P_1 and P_2 , are determined as shown in the figure below. This is accomplished using offset vectors, V_{front} and V_{side} , from the vehicle's center, C , which corresponds with its position as updated by Entity Service. Offset vectors for each vehicle type are defined by the implementor in the configuration file (see Section 9.3.2.1) and can be placed anywhere along the vehicle's track. Current implementations have placed these points toward the front of the vehicle to help compensate for simulation lag. If the points were located at the back of the vehicle, the track depressions appear to lag the vehicle's movement.



1. Currently, the number of Track DTR loop cycles before item deletion is set to $\text{TIMEOUT} \times \text{Entity Service update rate}$ where TIMEOUT is 12 (seconds) and the default update rate is 2 (Hz). After 24 iterations through the Track DTR, an item is removed. This is a minimum of 12 seconds. However, Entity Service also has its own timeout to remove entities from its list. Thus, an inactive entity will timeout for Entity Service first followed by the Track DTR.

Figure 46. Vehicle Instantaneous Track Points

The track point locations are set using the offset vectors prior to a simulation exercise. During a simulation, the instantaneous track points are computed using the rotation matrix M

$$P1 = M(V_{front}) + M(V_{side}) + C \quad (63)$$

and

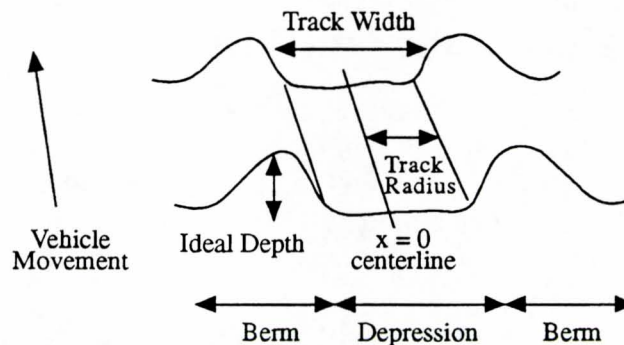
$$P2 = M(V_{front}) - M(V_{side}) + C \quad (64)$$

The vectors V_{front} and V_{side} can be pre-added so that only a single matrix multiplication is required. The current implementation ignores pitch and roll values of the vehicle orientation.

9.2.2 Soil Displacement

A kinematic heuristic is used to create a soil displacement that is visually effective. The track profile consists of depression and berm components as depicted in the figure below. Several parameters are adjusted to achieve the desired appearance. The following parameters are configurable by the user:

- Ideal Depth* - This value is the kinematic depth of the track and does not take into account the strength of the soil or the weight of the vehicle. Future work could make this depth a function of soil and vehicle parameters.
- Track Width* - This is the width of the depression created by a single vehicle track.
- Track Radius* - This value of half the width is used in most calculations.
- Berm Width* - This is the width of the berms on either side of the depression.



0703-6859

Figure 47. Track Profile and Parameters

The terrain patch affected by a vehicle track is determined from the track profile parameters listed above and the offset vectors described in the previous section. A query for the selected area is made to Terrain Service before applying the heuristic to the gridded elevation¹ values returned by the query.

Since the radius of the track, instead of its width, is often used in the calculations,

$$TrackRadius = \frac{TrackWidth}{2} \quad (65)$$

the change in the elevation attribute is computed as an offset. This offset is added to the sampled

1. Currently only the elevation attribute is modified; however, the model is easily extendible to other terrain attributes.

terrain data. The relation is expressed as:

$$\text{Offset} = \text{DepressionComponent} + \text{BermComponent} \quad (66)$$

where either berm or depression component is active for a particular grid or cell. Therefore, as the track profile is stepped across in direction x , the offset while within the track width is negative:

$$\text{DepressionComponent}(x) = -\text{IdealDepth} \quad (67)$$

for

$$-\text{TrackRadius} < x < \text{TrackRadius} \quad (68)$$

while the offset within berm widths is positive. The berm component changes in height from a maximum of *Track Depth* dropping linearly to zero at *Berm Width*. The descent to zero is moderated by the *Berm Slope* where larger slope values cause the berm to descend more rapidly. The *BermComponent* is expanded as:

$$\text{BermComponent}(x) = \left[1 - \left(\frac{x - \text{TrackRadius}}{\text{BermWidth}} \right) \right] \text{IdealDepth} \quad (69)$$

for cell values outside of the depression limits and within the berm limits

$$\text{TrackRadius} < x < (\text{TrackRadius} + \text{BermWidth}) . \quad (70)$$

Cells within the terrain patch that are outside of either berm or depression limits receive an offset of zero; thus remaining unchanged.

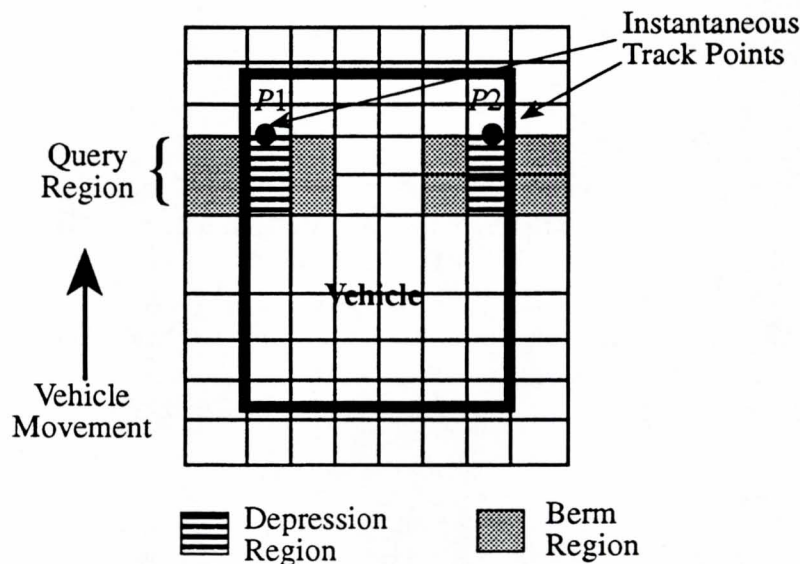
Track parameters are the same for all instantaneous track points on a particular vehicle. Consequently, each track on a vehicle has the same ideal depth and profile components. This does not mean that the ground beneath each track point will look the same, since elevation can vary from one track to the next.

9.2.3 Terrain Patch Query and Update

The terrain patch query takes advantage of the flexibility of the Terrain Service query mechanism. The offset has been described for a single row of the patch query. Subsequent rows receive the same offsets, as a function of x , as the first row. The current implementation queries Terrain Service for 2 rows as this is the minimum required by Terrain Service in order to make changes to the global database.

Terrain Service supports queries which can be aligned with the vehicle body. Therefore, the local entity coordinates are lined up with the tracks. The shape of the query extracted from the Terrain Service is depicted in the figure below. Since a consistent profile around the vehicle is needed, the points which are raised or lowered by the track algorithm will have a fixed position in the query buffer. During each cycle of the Track DTR, the Terrain Service query is similar. Changes are made to only a few locations in the query buffer which correspond to the instantaneous track

points $P1$ and $P2$.



0703-6860

Figure 48. Terrain Service query relative to the vehicle

Consider a database with resolution of 1 meter. A tank travelling at 40 miles/hour corresponds to a speed of approximately 7 meters/second. An update rate of at least 7 Hz is required to account for all of the ground covered. If laying track upon receipt of each entity state update does not produce a realistic display, the resolution of the query in the direction of vehicle movement, low resolution, can be enlarged to account for the ground covered between updates. Currently, this is accomplished by setting the y resolution in the configuration file. This resolution for the Terrain Service query remains constant throughout a simulation. Since the vehicle's previous location is already stored, the distance travelled could be used to affect the y resolution of the query. This results in a query of varying sizes depending upon the speed of the vehicle. This approach is also available (see Section 9.3.4 Programmer's Guide).

The converse of the speed problem is one of slow or no vehicle movement. If the terrain is modified with each entity state update, a stationary vehicle could sink. Thus, the algorithm checks each vehicle update to insure it has moved a minimum distance before processing the soil beneath it. This minimum distance is at least one cell resolution.

9.3 Implementation

9.3.1 Problem Statement

The track-making algorithm described above relies on DT Services to accomplish its task. When it is obtaining land vehicle position updates, it functions as a client to Entity Service using of the Entity Service Client Interface. When it is obtaining terrain patch beneath the vehicle, it functions as a client to Terrain Service using the Terrain Service Client Interface. This interface to Terrain Service also provides the capability of updating a terrain patch which is then integrated into the

global database by Terrain Service. Thus, the Track DTR provides the terrain changing algorithm and is shielded from the tasks of listening to the network for incoming PDUs, dead-reckoning vehicles, and interpolating gridded terrain.

9.3.2 User's Guide

Two implementations are possible with the track model -- alone and in conjunction with DT Services. The trackClient provides an interface to the track model much as the Entity Service Client Interface provides a link to Entity Service. The trackClient works in conjunction with both Entity and Terrain Service. This is accomplished by setting various track parameters in the configuration file prior to a simulation. Once Entity and Terrain Service are instantiated during the simulation, the trackClient is invoked which instantiates Entity and Terrain Service Client Interfaces and the VehicleTrackResource. The implementor does not have to create any code and simply edits the configuration file.

Alternately, the track model, or VehicleTrackResource, can be used by itself. High fidelity simulations performing many queries to Terrain Service should generate their own tracks to enhance coupling between vehicle states, velocities, and frequent terrain interactions. This requires colocation of the track model and the Vehicle simulation on a machine, probably within the same simulation. The trackClient which communicates with both Entity and Terrain Service is not necessary since the entity itself will be creating its tracks. The vehicle simulator simply makes use of the VehicleTrackResource public methods, querying and updating Terrain Service when feasible. This implementation requires the implementor to write code.

9.3.2.1 Configuration File

The Track DTR configuration file should be edited prior to a simulation. The track profile parameters described in Section 9.2.2 are specified in this file as well as terrain query resolution and vehicles of interest. The line consisting of 'TRACK_CONFIG_FILE' should appear as the first line in the file, otherwise the file is ignored. A '#' character at the beginning of a line marks a comment line which is ignored.

Data sets are used to specify different track profile parameters. Each data set specifies berm and depression widths for a track. These measurements are the same for all vehicles specified within this data set. A different set of track parameters is specified in subsequent data sets. Currently, a maximum of 5 data sets with a maximum of 3 vehicle types with a maximum of 2 tracks can be specified in the file.¹ Vehicle types are defined in the simulation entity identification file (see Section 9.3.2.2).

The layout of the sample file should be followed rather closely since the initialization process can be easily tricked by an incorrect configuration file. A sample configuration file follows:

TRACK_CONFIG_FILE

1. These limits can be changed by redefining the values in VehicleTrackResourceDefs.h++ and recompiling.

Set database query x and y resolution.

X resolution is limited from 0.1 to 5 and corresponds to the resolution
across the track profile along berm and depression components.

Y resolution is limited from 0.1 to 5 and corresponds to the direction of
movement of the vehicle.

xResolution 1.0

yResolution 1.0

Vehicle track data for use with Track DTR.

(Units are consistent with those of the vehicle simulation

parameters - meters for DIS.)

Total of vehicle track data sets represented. Each data set specifies berm
and depression widths for the tracks. These measurements are the same
used for all vehicles specified within this dataset. A different set of track
parameters are specified in a different dataset.

There are 2 sets of track parameters for this simulation.

datasetTotal 2

Modifiable database attributes. Currently only elevation attributes
can be modified.

0 = elevation

attribute 0

===== Data set 1 =====

Number of vehicle types represented by set 1 track parameters.

totalVehicles 1

List IDs for vehicle types represented by this data.

DIS Category: Tank

vehicleID TANK

#Number of instantaneous track points

trackPoints 2

#Track parameters

#Track Point 1 location: right of center

sideOffset 2.2

forwardOffset 0.0

#Track Point 2 location: left of center

sideOffset -2.2

forwardOffset 0.0

#berm width < depression width

depressionWidth 1.2

bermWidth 1.0

idealDepth 1.0

===== Data set 2=====

totalVehicles 1

List IDs for vehicle types represented by this data.

vehicleID BREACHER

trackPoints 2

#Track Point 1 location: right of center

sideOffset 2.2

forwardOffset 0.0

#Track Point 2 location: left of center

sideOffset -2.2

forwardOffset 0.0

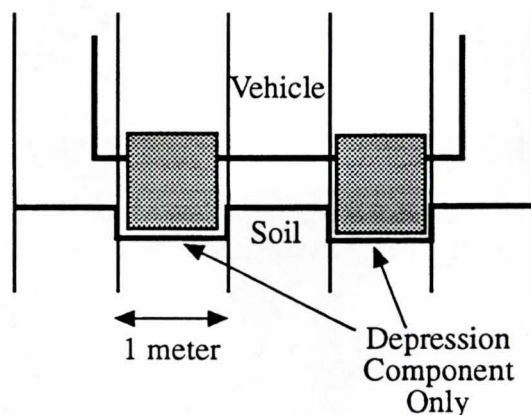
#no berm component

depressionWidth 1.0

bermWidth 0.0

idealDepth 1.0

Terrain resolution should be considered prior to applying both depression and berm components. The M1 track width is 0.5 meter with a 1.5 meter separation between the inner side of the tracks. If the terrain resolution is 1 meter, the depression component could be centered around the instantaneous track point without creating the berm components on either side as shown below. This option is invoked by setting the berm width to zero.



0703-6861

Figure 49. Depression Component (Only) Used for Tracks

9.3.2.2 Simulation Entity Identification File

Entity identification data is included with updates from Entity Service. This identification has been enumerated in the DIS standard. Some land vehicles which are appropriate track-makers are defined in the simulation entity identification file as entityID.h.¹ The vehicle types declared in the configuration file must be listed in this file for the track-maker to operate. Thus, if there is a vehicle of interest that is not specified in this file, make an addition. Each entry in the table must contain the following values.

- *Name*. Name of vehicle in quotes. Maximum length is 25 characters. The vehicleID used in the configuration file must match one of these names.
- *Kind*. Entity kind as described in the DIS standard.
- *Domain*. Entity domain as described in the standard.
- *Country*. Entity country as defined in the standard.
- *Category*. Entity category as defined in the standard.
- *Sub-category*. Entity sub-category as defined in the standard.
- *Specific*. Specific entity as described in the standard.
- *Extra*. Extra information which is typically ignored.

The Track DTR lays tracks for all vehicles that meet values specified. A value of zero is ignored and all comparisons will pass. The following is any excerpt from the file:

1. The file *ent_code.h* contains defined values for most DIS entities. These defines can be used to describe vehicles in the simulation entity file.

"TANK", PLATFORM, LAND, USA, TANK, 0, 0, 0

"BREACHER", PLATFORM, LAND, USA,
ARMORED_UTIL_VEHICLE, M548,0, 0

Most entries will have sub-category, specific, and extra values set to zero. This is because most vehicles can be adequately specified by the upper categories. The above specification for tank indicates all entities in the tank category, while the breacher (See "Introduction to Mobility and Vehicle Dynamics" on page 273.) is more specific.¹

9.3.3 Software Design

The Track DTR is implemented in a two tier hierarchy in which the `trackClient` provides an interface to the underlying model, the `VehicleTrackResource`. The track model is implemented in the `VehicleTrackResource` class.



Figure 50. Booch diagram for track model

The `VehicleTrackResource`, `EntityServiceClientInterface`, and `DatabaseServiceClientInterface` are instantiated upon execution of the `trackClient`. `Dictionary`, `ActiveVehicleType` and `ActiveVehicleKeyType` are also instantiated; these classes are associated with the bookkeeping of simulation vehicles. These object oriented tools are the dictionary, element and key, respectively, for a dictionary of vehicle types.

9.3.4 Programmer's Guide

```
class VehicleTrackResource
{
```

```
    protected:
```

```
        unsigned char  total_types,           //total of vehicles
                    number_of_attributes,    //total attributes to be modified
                    dataset_total,           //number of datasets
                    index,                   //dataset index
                    no_berm_flag,            //indicates no berm component
                    elev_flag;               //indicates elevation attribute to be modified
```

1. These values for the breacher were created by the DT project since there is not yet an enumeration in the DIS standard for this vehicle.

```

float      s_spacing,          //track profile spacing, x direction
           t_spacing;          //patch row spacing, y direction

trackData*  track_table;       //track offsets for each dataset

elevData*   elev_table;        //track profile parameters for each dataset

// Vehicle types and associated dataset table number.

applicableVehicleType applicable_vehicle_table[MAX_TOTAL_VEHICLES];
unsigned char vehicleDefined( char name[], EntityType* type );

void findAttributeQueryPatch(
                                trackData  track;
                                elevData   elev;
                                vector2D   query_patch[] );

void findQueryPatchesMinMax(
                                vector2D   (**attribute_extents)[4],
                                vector2D   (*query_extents)[4]);

double closestTrackPoint2D( unsigned char number_of_track_points,
                             vector2D origin,
                             vector2D points[]);

public:

VehicleTrackResource( char*      config_file,
                     unsigned int* err );

~VehicleTrackResource();

void dumpTables();

unsigned char vehicleOfInterest( EntityType es_type );

unsigned char vehicleMoved( vector2D current_location,
                           vector2D past_location,
                           vector2D* distance);

void findQueryPatch( vector2D entity_center,
                     Angles  entity_orientation,
                     int*    number_of_rows,
                     int*    number_of_columns,
                     vector2D world_patch_extents[]);

```

```

void localToWorldCoordinates2D( unsigned char number_of_points,
                                vector2D      center_point,
                                float          heading,
                                vector2D      offset_from_center_point,
                                vector2D      rotated_point[]);

void modifyElevation( unsigned char t_number_of_points,
                      unsigned char s_number_of_points,
                      vector2D      entity_origin,
                      Angles        entity_orientation,
                      float          attribute_values[][3]);

void modifyElevation( unsigned char t_number_of_points,
                      unsigned char s_number_of_points,
                      vector2D      entity_origin,
                      Angles        entity_orientation,
                      float          attribute_x[],
                      float          attribute_y[],
                      float          attribute_z[]);

unsigned char getNumberOfAttributes();

unsigned char getElevationFlag();

float getResolutionS();

float getResolutionT();

unsigned char getTrackPoints( vector2D points[] );

unsigned char getTableIndex();

};

```

The public methods within the VehicleTrackResource class are described below.¹

9.3.4.1 VehicleTrackResource(configFileName, error);

Vehicle Track Resource is the class constructor which uses information contained in the configuration file for initialization of track parameters and specifies vehicles of interest. A non-zero error indicates failure to properly initialize the Track DTR.

9.3.4.2 ~VehicleTrackResource();

This is the class destructor.

1. Several of the methods are not currently used.

9.3.4.3 void dumpTables();

This prints information pertaining to the class and input via the configuration file.

9.3.4.4 unsigned char vehicleOfInterest(EType);

Given a vehicle's identification, this returns a 1 if the vehicle is one for which tracks will be made.

9.3.4.5 unsigned char vehicleMoved(current_location, past_location, distance);

This returns a 1 if the vehicle has moved at least the resolution of a terrain cell. It also calculates the distance travelled since the previous update.

9.3.4.6 void findQueryPatch(entity_center, entity_orientation, number_of_rows, number_of_columns, world_patch_extents);

This determines the patch bounding points and the number of rows and columns for the Terrain Service query.

9.3.4.7 void localToWorldCoordinates2D(number_of_points, center_point, heading, offset_from_center_point, rotated_point);

This converts local terrain patch offsets to world database coordinates for use in the patch query to Terrain Service. This method uses heading only and is currently used with the Track DTR.

9.3.4.8 void modifyElevation(t_number_of_points, s_number_of_points, entity_origin, entity_orientation, attribute_s, attribute_y, attribute_z);

This modifies an array of elevation values for the terrain patch based on the track-making algorithm. This is used with Terrain Service client applications.

9.3.4.9 void modifyElevation(t_number_of_points, s_number_of_points, entity_origin, entity_orientation, attribute_values);

This modifies an 3D array of interlaced elevation values for the terrain patch based on the track-making algorithm. This is used with DTDB client applications.

9.3.4.10 unsigned char getNumberOfAttributes();

This returns the number of terrain attributes which are being effected by the Track DTR. Cur-

rently this value is one since elevation is the only attribute being modified. Future extensions to the model could involve soil strength or thermal attribute layers.

9.3.4.11 unsigned char getElevationFlag();

This returns 1 if the elevation attribute will be modified by the Track DTR. Currently, this is always true for the Track DTR.

9.3.4.12 float getResolutionS();

This returns the resolution across the track profile (see x in Section 9.2.2 .)

9.3.4.13 float getResolutionT();

This returns the resolution of the terrain patch rows (see Section 9.2.3).

9.3.4.14 void getTrackPoints(numberOfPoints, points);

This returns the number of instantaneous track points and their offset values (see Section 9.2.1).

9.3.4.15 unsigned char getTableIndex();

This returns the index of the table containing track parameter information for a particular vehicle

10.0 Thermal DTR

10.1 Introduction to Thermal DTR

10.1.1 Background

The Thermal DTR was developed to simulate the natural cooling of terrain due to ambient temperature conditions. Terrain temperature is important in many simulations. The dynamic terrain database has the ability to store temperature distribution across a database. When a participant in a simulation changes this temperature (due to detonation, vehicle passage, etc.) they are not responsible for keeping the temperature current. Once the temperature of the terrain has been modified, the simulator that modified it does not have to keep track of what was modified, or model the terrain cooling over time. This is the job of the thermal DTR. Whenever terrain temperature has been changed from ambient, the thermal DTR will gradually converge the database temperature towards the ambient temperature. This DTR is similar in focus to the Soil DTR that models the soil flow as a reaction to geometry changes.

10.1.2 Problem Statement

The fundamental requirement of the Thermal DTR is that it must be capable of gradually cooling the terrain until it reaches ambient temperature (or very close to ambient temperature). This must be done as a passive (not interactive) process. In other words, it should only react to temperature changes made by other programs and not initiate any action on its own.

10.1.3 Solution

The thermal DTR is a simple implementation designed to cool the terrain without too much overhead or process effect on other parts of the shared simulation. The DTR simply passes over all of the terrain one patch at a time. It reads the temperature of that patch queried from Terrain Service, passes the patch through a cooling model, and returns the modified patch to Terrain Service. By gradually passing across the entire database, it slowly cools the terrain. Although the capacity for cooling models of varying fidelity exist, only a simple model has been implemented. This model simply takes the difference between terrain temperature and ambient temperature at each point and reduces the difference by a cooling percentage. Although simple, this DTR provides a way to gradually cool the terrain without losing the general temperature distribution across the database.

10.1.4 Constraints/Assumptions

The only constraint on the Thermal DTR is access to the Terrain Service. Terrain Service and the Thermal DTR must run on the same machine.

10.2 Analysis

Initial design considered tracking the terrain hot spots. However, this proved to be too much overhead for a such a simple DTR so the passover approach was implemented. What this basically does is gradually scan the entire database, one patch at a time, cooling each patch as it comes and then moves on to the next patch. This frees the DTR from having to keep track of all of the host spots on the terrain, it simply cools all of it. Since the DTR runs slowly, this doesn't require too many updates.

10.3 Implementation

10.3.1 Problem Statement

The premise of the Thermal DTR was to provide a simple tool that could be run in the background of a simulation and provide basic cooling effects. Since temperature can be an important part of a simulation, it is important to model the behavior of the environment as regards to passive cooling of the terrain.

10.3.2 User's Guide

Since the Thermal DTR is a reactive process, it does not require a user interface. It is simply started and left running for the duration of a shared simulation. To run the Thermal DTR, the user must set up three configuration files which are described below. The Thermal DTR is invoked with the main configuration file as it's only argument. The Terrain Service must be run on the same machine for the Thermal DTR to function.

10.3.2.1 Configuration Files

As mentioned, there is one main configuration file for the Thermal DTR. This file specifies the other configuration files used to setup the Terrain Service Client Interface and the thermal model file. A sample main configuration file is shown below.

```
# The Thermal DTR's standard configuration file.

# The name of Terrain Portal's standard configuration file.
terrain_portal_file terrain_portal.cfg

# The name of Thermal Model's standard configuration file.
thermal_model_file thermal_model.cfg
```

The standard terrain network file currently contains the logon information that is required to connect to the Terrain Service. Without this information, the program cannot successfully logon to either service. This files is listed below.

```

# The standard configuration file for Terrain_Portal.

# The identifying name used to confirm logons.
application_name Thermal_DTR

# An estimation of how frequently updates will be requested.
interface_cycle_rate 1

# The shared memory area used to initially connect to the service.
control_shared_memory_key 0x00005000

```

The last file specified in the master configuration file is the thermal model's standard configuration file. This file specifies the extents of the database, the ambient temperature, and the cooling rate.

```

# The Thermal Model's standard configuration file.

# The number of columns in the cooling patch
number_of_columns_per_patch 16

# The number of rows in the cooling patch
number_of_rows_per_patch 16

# The number of columns of cooling patches
number_of_patch_columns 16

# The number of rows of cooling patches
number_of_patch_rows 16

# The X-value of the south west corner of the database
south_west_X 0.0f

# The Y-value of the south west corner of the database
south_west_Y 0.0f

# The X-value of the north east corner of the database
north_east_X 256.0f

# The Y-value of the north east corner of the database
north_east_Y 256.0f

# The ambient temperature of the database
ambient_temperature 25.0f

# The percentage rate at which the database
# cools during a simulation cycle

```

cooling_rate 0.1f

The 'f' following some of the numbers forces the compiler to use 4-byte floats rather than 8-byte doubles to represent these constants.

10.3.3 Software Design

The only new class developed for the Thermal DTR is the Thermal Model Class. The purpose of this class is to provide a form of cooling for the simulated environment. It is mainly a placeholder for future, more realistic, approaches to cooling.

This class begins by dividing the terrain into a set of patches, with the number of rows and columns of these patches determined by the thermal model configuration file. Each patch is then divided into a set of elements, or location values, with the number of rows and columns also given in the configuration file. To cool the terrain, the thermal model simply goes through each of the patches, one per simulation cycle, queries Terrain Service for the temperatures at the patch locations, cools each temperature value, and updates Terrain Service with the modified data. The actual cooling is very simple. The ambient temperature of the ground is read from the configuration file, as is the cooling rate. To cool a point temperature, the difference between the actual temperature and ambient is calculated. This difference is multiplied by the cooling rate to get the drop in temperature (or rise if the ground was actually too cool instead of too hot). This difference is then subtracted from the actual temperature to get the new temperature.

10.3.4 Programmer's Guide

Please refer to the SimHost programmers guide for references on most of the classes.

11.0 Minefield DTR

11.1 Introduction to Minefield DTR

11.1.1 Background

Minefields are critical to training for combat readiness and will provide DT a more realistic land battle environment. This section describes an approach to a generic Minefield Dynamic Terrain Resource (DTR) which uses the existing DT capabilities of cratering, soil slumping, Entity and Terrain Services. The software implementation of this approach is referred to as the Minefield DTR.

11.1.2 Problem Statement

A process to simulate minefields must have the ability to simulate multiple minefields as well as an arbitrary number of mines within each field. To increase the robustness of the process, it could maintain the boundary of each minefield to aid the intersection testing of each mine with each entity.

11.1.3 Solution

The solution is best described by the following example. The Minefield DTR keeps track of the mines that are specified in a configuration file. The Entity Service updates the Minefield DTR with various entities' locations. A tank enters the minefield. Entity Service continues to send a stream of information regarding the tank's location. If a tank is co-located with a mine, the Minefield DTR invokes a detonation. The Crater DTR picks up the detonation and announces a terrain change. The Terrain Service incorporates this terrain change into the database and announces terrain modifications to its clients.

11.1.4 Constraints/Assumptions

The minefield model operates under the following constraints and assumptions:

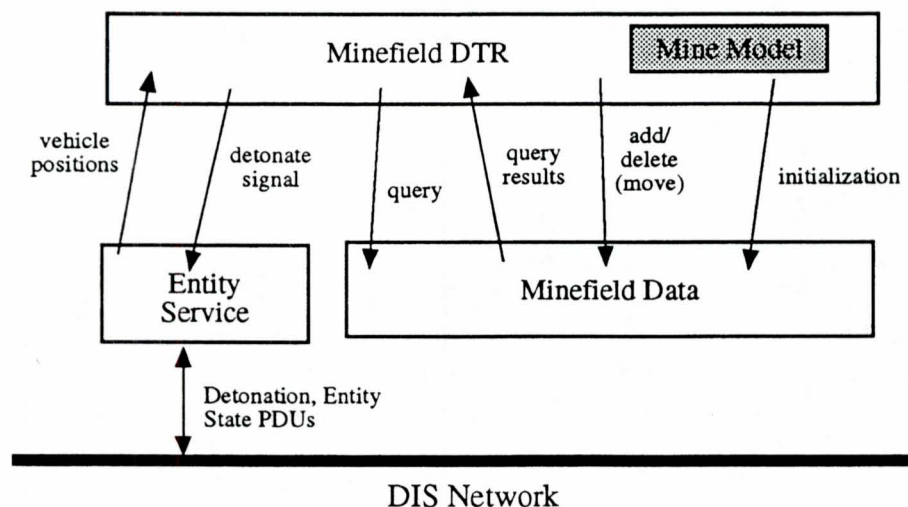
- The Minefield DTR is a client to the Entity Service which provides the model with vehicle updates.
- The generic minefield sends a detonation when one of its members is triggered. Triggering according to vehicle proximity is used in this implementation. Triggering methods such as acoustic, seismic, pressure, magnetic, and IR could be future extensions.
- This implementation will consider autonomous mines. The trigger will be co-located with the mine.
- Emplacement of mines occurs prior to the simulation. This data is contained in a configuration file. Later enhancements to the minefield could include emplacement control mechanisms such as conventional burying and advanced scattering methods.

11.2 Analysis

11.2.1 Minefield Model

Minefields can be of arbitrary extent within the scope of the terrain database dimensions. The minefields are checked as vehicles move around the system. The Minefield DTR keeps continuous watch for mine-vehicle meetings, and notifies the Entity Service of a detonation. This results in the sending of a detonation PDU.

Following a mine-vehicle encounter, a detonation PDU can be interpreted by the offending vehicle to update its status. The Minefield DTR removes the mine.¹ When a detonation impacts the terrain, a crater is created by the Crater DTR (See "Introduction to Crater DTR" on page 113.). These craters resulting from explosion of mines change the terrain profile. Smoothing of the modified terrain surface is accomplished by the Soil DTR (See "Introduction to Soil DTR" on page 81.).



0703-7047

Figure 51. Minefield Two-tiered Approach

The Minefield DTR follows a two-tiered approach as shown above. The top layer, Entity Service, is connected to the DIS network while the bottom layer focuses on the minefield solution. The Minefield DTR communicates with the Entity Service through the EntityServiceClientInterface. The Entity Service provides a list of dead-reckoned entities and their positions to its clients.

The Minefield DTR maintains a list of active entities whose location is compared with the locations of mines in the minefield data structure to determine if any are close enough for detonation. In this case, the Minefield DTR sends detonation information to the Entity Service which issues the Detonation PDU. The destroyed mine is then removed from the minefield data structure.

1. Future enhancements could allow the Minefield DTR to mediate the clearing of minefields.

11.3 Implementation

11.3.1 Problem Statement

The minefield algorithm described above relies on the Entity Service to accomplish its task. As a client to the Entity Service, it obtains vehicle updates using the `EntityServiceClientInterface`.

11.3.2 User's Guide

11.3.2.1 Configuration File

The Minefield DTR configuration file should be edited prior to a simulation. A configuration file, named **mines.dat**, is used to input the location of the mines.¹ In this simplified approach, two different parameters are used in the configuration file as follows:

- field n
where n is the number of mines in the current minefield
- mine $x\ y\ z$
where x , y and z specify the location of a mine²

For example, the following is a simple minefield configuration file where the first line containing `MINE_FIELD_DATA_FILE` is required.

```
MINEFIELD_DATA_FILE

field 12
mine 190.0 100.0 400.0
mine 50.0 15.0 400.0
mine 70.0 20.0 400.0
mine 150.0 100.0 400.0
mine 130.0 80.0 400.0
mine 90.0 60.0 400.0
mine 130.0 100.0 400.0
mine 490.0 250.0 400.0
mine 390.0 50.0 400.0
mine 400.0 200.0 400.0
mine 590.0 270.0 400.0
mine 890.0 100.0 400.0

field 3
mine 20.0 20.0 400.0
mine 25.0 20.0 400.0
mine 20.0 25.0 400.0
```

-
1. Future enhancements could allow other mechanisms of mine delivery such as a vehicle deploying mines.
 2. z is currently not used in the intersection testing.

11.3.2.2 Software Design

The figure below shows the Booch diagram of classes used in the Minefield DTR.

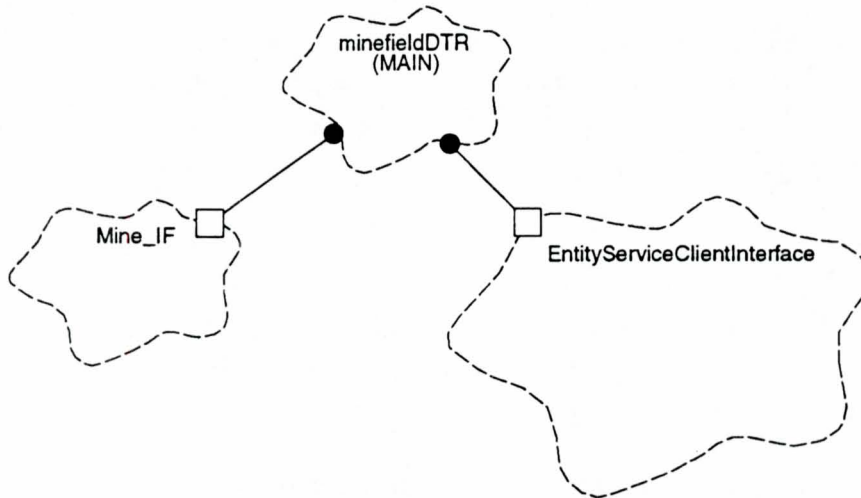


Figure 52. Booch diagram of the Minefield DTR

11.3.3 Programmer's Guide

The public members of the class specification for the mine model class follows:

```
class MinesIF
{
    public:
        // Constructor. Reads data file of mine locations
        MinesIF();

        // Destructor.
        ~MinesIF();

        // Returns whether or not the vehicle's position (x,y) is
        // in a mine radius
        int enteredBy(float x, float y);
};
```

The constructor simply reads the configuration file that contains the list of mines. The enteredBy() method returns whether or not the vehicle position specified by (x, y) is within a mine. If so, it returns TRUE and the mine is deleted from the minefield. Otherwise, it returns FALSE.

Note that the minefieldDTR itself must concern itself with the domain of the vehicles themselves. Air vehicles such as an F18 should not detonate mines. Therefore, the Minefield DTR does not check non-land vehicles against the mines.

More Shared Environment Clients

The components introduced in this section are also clients to Dynamic Terrain's shared environment (See "Dynamic Terrain Shared Environment" on page 1.); however, they are not DTRs.

Programs that make use of the Shared Environment are collectively known as Client Applications. Client Applications are responsible for simulating some element of the shared virtual space. This element can be either a traditional entity or the simulation of some part of the environment. Client Applications typically run on a single physical machine; however, in some cases they may be composed of several separate processes.

The broad category of Client Applications is further broken down into DTRs and other client applications. Other client applications include traditional vehicle simulations, image generators, and other tools for visualization.

12.0 Visualizer

12.1 Introduction To Visualizer

12.1.1 Background

A typical image generator produces a single picture of the terrain that incorporates all of its supported terrain attributes. For instance, a "soil type" attribute is used to decide how to represent the terrain (such as the color of the polygons or the texture placed on the polygons). Early in the Dynamic Terrain (DT) project, it was evident that a program to view individual dynamic attributes would be useful. The Visualizer is that program.

The Visualizer is actually one of the older pieces of the DT system. It was originally written as a passive viewer (i.e. not connected to any network) of the DTPatch system developed in Architecture 1 [39]. As the project continued, it was modified to use the Architecture 3 Dynamic Terrain Database (DTDB), but it remained passive. Once the Entity Service and Terrain Service were developed, the Visualizer underwent another update to become an active viewer, reading and visualizing entities and dynamic terrain changes from the network. For more information, please refer to the Entity Service and Terrain Service documents.

12.1.2 Problem Statement

The Visualizer was developed to provide a means to show the individual attributes for a terrain database. The user can see each attribute itself as well as the changes made to each one as it occurs. To aid in this effort, a user-friendly interface needs to exist to allow the user the most flexibility to access the database while remaining relatively easy to use. Furthermore, one of the more recent additions has been the visualization of dynamic fluids as a client application of the Fluid Service. For more information, refer to the Fluid Service document.

12.1.3 Solution

The Visualizer displays each attribute as a plane in the graphics display. For elevations, an actual elevation mesh is displayed while a simple color scaling is used for the other attributes. As terrain attributes are received from the network, the display is updated to show the new changes. Furthermore, entities are displayed through the use of a symbol database or a simple ASCII text string, selected by the user at run-time.

To solve the interface problem, a user interface was developed using the FORMS Library [46]. This interface provides the user with the ability to move the eyepoint, change the bounding box of the terrain that is displayed, alter the database rotation, change the post spacing used for the attributes, turn the display of attributes to on, off or plant, and vary the spacing between attribute planes in the display. Furthermore, a 3-D cursor gives the user a method to cause detonations as well as query values in the attributes.

12.1.4 Constraints / Assumptions

The only constraint on the Visualizer is that it must run on a Silicon Graphics machine (since it is dependent on the Graphics Library (GL) and Forms).

12.2 Implementation

12.2.1 Problem Statement

This section explains the interaction between the Visualizer and the DIS network. Before going further, note that the Visualizer uses the Entity Service, Terrain Service and Fluid Service in order to receive PDUs. A detailed discussion of these programs is beyond the scope of this document, but the reader should be aware that both services are based on synchronous reading of "messages" based on PDUs received.

When each Entity State Message is received from the Entity Service, the entity ID contained within it is checked against all current entities. If it is not found, the Visualizer assumes that a new entity has been created. Whether found or not, the Visualizer will store the DIS type and orientation data of the entity for use in updating the display. After all new states for entities have been read, the entities are checked to see if they were updated. Currently, if any entity has not been updated from the Entity Service after three rounds of updating, the entity is considered to have timed-out and is deleted.

The Visualizer does not react to incoming Detonation Messages. However, if the user clicks the left button while moving the 3D cursor, the Visualizer will write a Detonation Message to the Entity Service at the cursor's location. *NOTE: this Detonation Message will not contain valid values for all fields since there is no entity associated with the detonation.* The exact location of the detonation actually varies slightly (± 5 units) for an added visual effect on image generators. Currently, only one munition DIS type is supported and it is (2, 9, 222, 1, 1, 0, 0). The munition type (in the burst descriptor) and the location are the only non-header fields properly set by the Visualizer. The rest are simply initialized to zero.

When an Attribute Message is received, the Visualizer uses the bounding box of the update within the message to find an axis-aligned bounding box. The new bounding box is used to requery the database at the current spacing and the display is redrawn. The Visualizer itself cannot change the terrain at this time, but adding this feature is an aspect that has been discussed.

Similarly, when a Fluid Message is received, the Visualizer uses the fluid boundary list contained in the message to draw the fluid as solid blue polygons. Because the Fluid Service is in its first iteration and has not been optimized yet, the entire fluid database is queried and returned when it has changed.

12.2.2 User's Guide

Before starting the Visualizer, a configuration file must first be created. The configuration file for

the Visualizer has the following parameters:

FrameRate n

where n is the number of times per second that the Visualizer should read data from the Entity Service and check for updates from the Terrain Service.

postSpacingX $s1$

postSpacingY $s1$

where $s1$ and $s2$ is the spacing in the X and Y directions, respectively, at start-up.

rotationX $r1$

rotationY $r2$

rotationZ $r3$

where $r1$, $r2$ and $r3$ are the starting viewing rotation about the X, Y and Z-axis respectively.

eyeX x

eyeY y

eyeZ z

where x , y and z are the starting eyepoint location.

eyeScale s

where s is the scale of movement of the eyepoint positioners on the Forms interface.

swX x

swY y

where x and y represent the location of the south-west point of the database in DTDB coordinates.

neX x

neY y

where x and y represent the location of the north-east point of the database in DTDB coordinates.

attribute i b $r1$ $g1$ $b1$ m $r2$ $g2$ $b2$ n

where i is the key for this attribute and n is a name for the attribute (which can include whitespace). b and m represent the range for this attribute. $(r1, g1, b1)$ represents the RGB color to use for a value of b , and $(r2, g2, b2)$ represents the color to use for a value of m . Colors for values between b and m are interpolated.

symbolFile f

where *f* is the filename of the symbol database.

```
DISmodel k d c cg scg s e n t
```

where *k*, *d*, *c*, *cg*, *scg*, *s* and *e* represent the kind, domain, country, category, subcategory, specific and extra components of the DIS entity types, *n* is the name of the symbol (in the symbol database) to use for this entity, and where *t* is the text name of the model (which can include whitespace). If an entity has a DIS type not defined by one of these parameters in the configuration file, then unexpected results may occur.

A simple configuration file might look like:

```
FrameRate 2

postSpacingX 2.0
postSpacingY 2.0

rotationX 0.0
rotationY 0.0
rotationZ 0.0

eyeX 56.0
eyeY 56.0
eyeZ 550.0

eyeScale 10.0

swX 0.0
swY 0.0
neX 112.0
neY 112.0

attribute 0 400.0 0 255 0 410.0 0 255 0 Elevations
attribute 1 350.0 50 0 0 460.0 255 0 0 Thermal Intensity

symbolFile test.sym

DISEntityType 1 1 225 1 1 1 0 m1 M1A1
DISEntityType 1 2 225 1 15 0 0 f18 F18
DISEntityType 1 1 225 3 6 0 0 dozer Dozer
```

Once a configuration file exists, run the Visualizer by placing the name of the configuration file as a command-line parameter to the Visualizer. For example, "vis vis.cfg". Figure 53 shows the control panel of the Visualizer's user interface.

12.2.3 Software Design

Figure 54 shows the Booch diagram of the classes of the Visualizer. As is evident in the figure, the

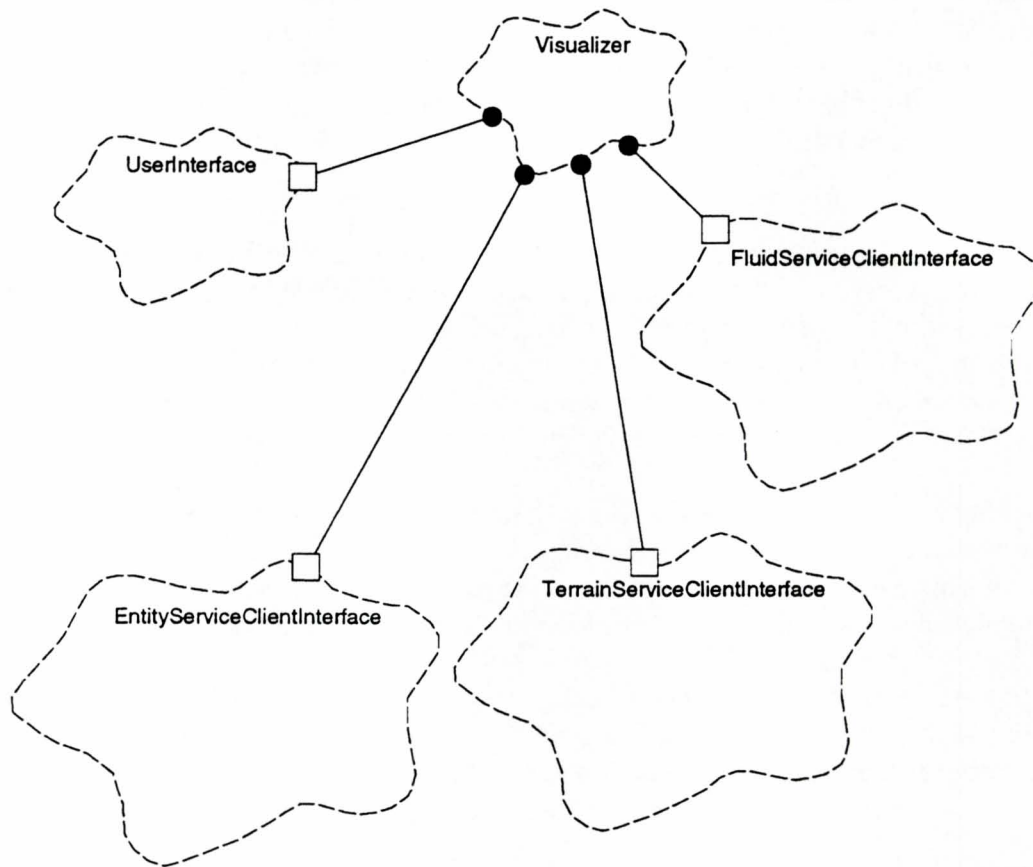


Figure 54. Booch diagram of Visualizer classes

Visualizer at this time does not depend on inheritance. Instead, the major division (the User Interface) has been split into a separate class that is used by the main Visualizer class. Depending on the desire to abstract the Visualizer, some sort of common superclass could be developed.

12.2.4 Programmer's Guide

Due to the nature of the Visualizer, it did not go through a significant design process like the Services and the IG/IG Host did. Therefore, there are only minor things that a programmer of an application like the Visualizer should know. At the beginning, one design decision was made was to keep the user interface in a separate class. This allows the user interface to be changed without

affecting the main Visualizer class (for example, we might switch from GL-based Forms to X-based XForms).

A view of the Visualizer class definition will show that only three non-constructor/destructor methods exist: `setTimerValues()`, `processUserInterface()`, and `checkServices()`. The first, `setTimerValues()`, exists to allow the main program to determine how to set the alarm for synchronized access to the Services. The main Visualizer is controlled by the latter two methods. `processUserInterface()` is called to check for user input from the interface (as well as to redraw the graphical display). `checkServices()` deals with queries from the Services, and is the more interesting of the two.

The redraw functionality of `processUserInterface()` is actually one of the less desirable parts of the code. In order to preserve the information regarding how the user interface worked, the drawing had to be done within the `UserInterface` class (since at some point in the future, the user interface may not exist on an Silicon Graphics machine running GL). This is not a major problem, but the programmer should be aware that a fair amount of data is passed to the user interface so that it can perform this function (since it does not and should not have a terrain database itself).

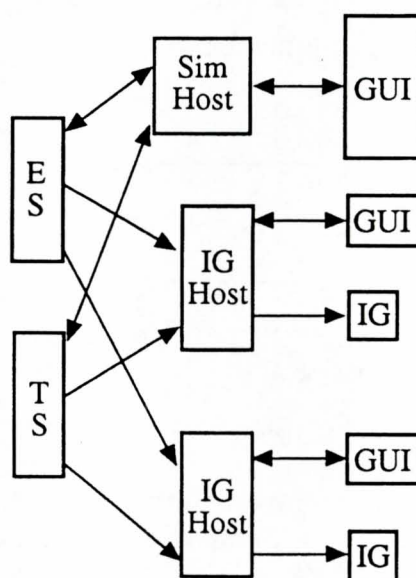
Within `checkServices()`, when the Visualizer is notified of a terrain change from the Terrain Service, it must then requery that part of the database, but at the resolution specified by the user in the user interface. This data will then be redrawn in the next simulation loop when `processUserInterface()` is called as discussed above. Updates to the fluid database are processed similarly.

Even though the Visualizer contains many functions and uses the Services extensively, it is not too complex and the programmer should have no difficulty in writing modifications. The only keys to remember is that communication with the Services is synchronized, and that a great deal of power exists within the HeyYou/Query/Update mechanism of the Terrain Service.

13.0 IG Host

13.1 Introduction to the IG Host

This document describes what each component of the overall simulation system encompasses including responsibilities, functions, and kinds of application programmer interface (API) libraries are used. The components of the simulation can be visualized by referring to the diagram shown below. This document is meant as a general overview of the system design



0703-6895

Figure 55. Simulation Design

components, and will further describe how the processes that the system employs work.

It is important to note here that all the code for this simulation was written C++ for Silicon Graphics machines, using IRIX 4.0.5x and 5.2x.

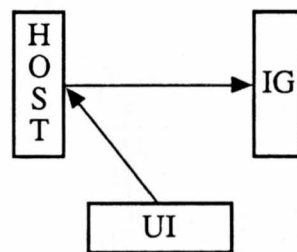
This work was intended as a redesign of the new Tracked Vehicle Simulator and was to implement the Model-View-Controller (MVC) framework proposed in "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80" by Glenn E. Krasner and Stephen T. Pope. This method seems to present the simplest, most powerful and versatile framework for simulation design. Any object or process can be modelled by the simple concept of MVC. Although the framework was created around the Smalltalk programming language, theoretically any object-oriented language should suffice. The reader should refer to the paper by Krasner and Pope for a detailed description of MVC.

A group of researchers at the U.S. Army Construction Engineering Research Laboratories have proposed an extension to Krasner and Pope's work. In "A Framework for Flight Vehicle Simulation" by Jeffrey Wallace, et al., the authors present a modified MVC framework for flight vehicle simulation. The programming language used is ModSim, a combination of Modula-2 and Small-

talk (and C added for good measure). The actual MVC design which is recommended will come from this extension, not from the original proposal by Krasner.

The design of the simulation host application and the image generator host has been covered in another document. However, the description of the design is briefly revisited here for easier understanding of how the graphical user interface (GUI) is included. The involvement of the graphical user interface may influence the design of the system quite heavily, however. This involvement was not included in the previous document's discussion.

A quick illustration of the classical view of how a simulator is configured may help in providing a starting point. The figure below illustrates the classical view of the connectivity between the host application, user interface, and the image generator.



0703-6896

Figure 56. Classical View of Host-IG-UI

Note that this figure implies a *non-graphical* user interface. A user interface in training simulators is usually considered to be strictly hardware. For a hardware interface, this configuration is quite adequate and provides the simplest, most powerful design. Most driver-trainer simulators do not have a GUI; all the controls resemble the real ones as much as possible.

However, we will investigate the implementation of a graphical user interface, as well as possible hardware interaction with the user for later implementation. Therefore, it is likely that the classical design will not work well. For now, we will only be concerned with a GUI for controlling the simulation application, and not with hardware controls.

13.2 Analysis

Dan Mapes has analyzed the paper by Wallace, et al. and has presented some comments through e-mail to various people in the Visual Systems Lab. He compares the MVC design as presented by Krasner to the extension by Wallace. His analysis is reproduced here:

MVC Analysis

Jeffrey Wallace from the U.S. Army Construction Engineering Research Labs proposes a framework for flight vehicle simulation using the Model-View-Controller (MVC) framework. It has been suggested as a good approach not only for flying vehicles but ALL DIS vehicles. I have studied his articles and MVC and can now offer some analysis of this approach.

The MVC can basically be seen as a stimulus response to the stimulus and propagation of change [Wallace84]. The stimulus can be from a controller, or input device, which has access to a models change methods. The response to a model change method would be the recalculation of the model. Once recalculated, the model notifies all views and controllers which are dependant on its state that a change has occurred. All dependant views and controllers then query the model for appropriate state information and then update themselves accordingly. It is important to note that the update and rendering process is driven by the controllers.

While this description of MVC has been over-generalized and many details have certainly been excluded, this description provides a general view of MVC which can be used to highlight various strengths and weaknesses. In the strengths category, the MVC provides a nice separation of responsibility and defines the interactions between each area of responsibility. The view can be generalized to simply be an object which has only a read interface to the model and allow views to be treated as controllers for other models. The propagation of results through a tree of models would result every time the controller at the root changed the root model. This "internal" update mechanism could be used to maintain a hierarchical tree of physical joints or relationships. Cycles in the joint dependencies could even be resolved through relaxation without an external solver being used. The primary problem that I have with this approach is due to the automatic updates.

Depending on the acceptable error, a physical simulations time intervals can vary. The update frequency of a sensor may depend on communication parameters and NOT error. In MVC, the sensors would drive the physical models update frequency and each physical model update would drive the framerate. In Wallace's article, he adds a "VM state strobe" which prompts the view to update on frame boundaries. This is NOT MVC, it is a modification to get around the problem between model update and frame update frequencies. There is no similar modification isolating sensor sampling and model update frequencies. This in itself is a weakness in the Wallace framework.

RE: models storing dependencies

The nature of real-time simulation requires that an entire frame be rendered at fixed intervals. In contrast, a windowing system only needs to be updated in the active pane and then only when a sensor invokes changes. While it is interesting to think about editing the Z-buffer instead of rebuilding it each frame, it is not really a practical option. Since the frame must be rebuilt, then the concept of having the VIEW update by referencing the MODEL at frame intervals triggered by a "strobe" is valid. Using this method however, an explicit dependency list would no longer need to be stored inside the model. The model could be abstracted as a pure source of state information as a function of time.

RE: decoupling frame, model & sensor update rates

The model could update whenever a state-time request is issued by a view. It could integrate at the required time step between the last request time to the current request time. Objects which similarly maintain a model of sensor state as a function of time can be queried at each step of the integration. The actual "Sensor" device can be queried by its own "strobe" set based on communication rates and results stored in queues.

RE: The value of dependency trees

The dependency trees which formed from the MVC approach not only updated all changed objects but defined the ORDER in which objects updated. This is still important when cycles of dependencies exist. Consider an example where a turret is dependant on the position of a tank. If the turret position is updated, then the tank position is updated and then the frame is rendered, the turret may not appear correctly relative to the tank. The

turret must update based on the current, not the last, position of the tank to be visually correct.

RE: Another possible approach

Here is an interesting MVC "like" framework to control the simulation by updating a simulation TIME model. A model of simulation time would store a dependency list of all time dependant objects (views). Each time dependant object would itself be some model of state with its own list of dependant models. Each time dependant object/model would access appropriate continuous time-state functions to update itself and then notify all of its dependant models to update. The only "controller" would be an object which uses a time event scheduler to generate a "strobe" for the next frame update cycle. The controller when strobed would simply change the current universe time in the model. The change in time then triggers the entire time dependant universe to be updated. When cycles exist and a relaxation cycle is started, dependencies to objects which are renderable could cause rendering side effects if rendering were tied to the update graph. Because of this, rendering should be done immediately following the model update. The frame buffer is "swapped" at the beginning of the next timing cycle.

This example follows the spirit of the classes proposed 4/11 with the addition of the MVC "time" model. Joints which defined a relationship of state between spaces would be time dependant models. Joints would be passed pointers to continuous space-time functions defining their remaining degrees of freedom. Agents would still be the mechanism to initiate function/behavior changes and would also be time dependant objects.

One problem with this approach would be attempting to resolve dependency cycles by solving constraint equations. Because the entire cycle must be considered, it must be identified and all objects within treated as a group. While this could certainly be done, keeping track of dynamically changing object cycle groups may be more bother than its worth. On a positive note however, any constraint solver would have to identify these groups.

Conclusion:

The MVC framework as specified in Smalltalk-80 is not a good framework to be used for simulation without modification. The Wallace article claims to use a MVC framework but has adapted it to the point where it is no longer recognizable as MVC. The aspect of MVC which allows control of view update order could be used effectively along with a universe TIME model to update state dependencies with cycles.

The nature of Mapes's work requires attention to details about the sensors in a simulation. Therefore, most of the arguments stem from this background. This is inherent to the nature of Virtual Environments, as the main goal of VE (also known as Virtual Reality, or VR) is to supplant the user's sensory input with the fabricated sensors on the computer. Therefore, the sensor functions in a simulation must be as real and as fast as possible.

We also need sensory input in our tracked vehicle simulation. It may even be beneficial to heed the advice of Dan Mapes in this aspect of our work. Our goal is not entirely an accurate simulation as if in a virtual environment, but it is something we should be aware of and not to close our doors to. If it happens to be a simple extension, then we can implement a VE for the tracked vehicles we simulate, using or building any appropriate hardware.

Another very important point by Mapes is the problem of the time model. ModSim was designed as a lightweight, multi-threaded environment to facilitate the framework proposed by Wallace. We will not be using ModSim, however. We will be using C++, a single-threaded, non-interpret-

tive programming language. Therefore, some of the concepts presented by Wallace are not easily implemented. Among these is the concept of *triggers*. The objects cannot run simultaneously and completely independently without an centralized "object" or "model" manager to handle the simulation. The ModSim "operating system" automatically handles the multi-threaded objects in the simulation. Asynchronous time steps among the objects is not possible. One object cannot be triggered by another object while performing its own simulation loop or adjusting to interruptions from external sources. While it may be possible to implement this in C++, the language does not inherently provide this type of control. Complicated shared memory structures and multiple processes would need to be implemented.

It is relevant to point out, however, that Wallace is justified in exploiting any of the advantages of ModSim for the flight vehicle simulation. Likewise, we would exploit any benefits gleaned from a C++ environment (not the least of which is code reuse). Wallace's framework does provide a simulation foundation which we may adopt for the tracked vehicle simulator. I say this despite Mapes's arguments above.

13.3 TVS Decisions

It seems appropriate to reflect upon different decisions that were taken during the creation of the Tracked Vehicle Simulator (TVS). These decisions, in one way or another, contribute to the design issues for the IG Host, Simulation Host, and even the IG. Some are basic, but their importance cannot be overlooked. For instance, it seems obvious enough by now that object-oriented design was chosen for the DT software. The rationale behind the decision to use OOPS and C++ in particular are given below.

13.3.1 Performer was used for Flight Files.

The reason for this should be obvious. Performer was designed by Silicon Graphics (SGI) to provide the most powerful simulation environment possible on their systems. It seems unlikely that we could implement a better system on our own. Furthermore, Software Systems has worked with SGI to incorporate a MultiGen Flight loader in their Performer software. This is especially useful for the new versions of Flight files which will have proprietary formats.

13.3.2 The TVS separated into managers for each major function

This was our primary attempt at an object-oriented design. We viewed encapsulation of the controlling managers as a valid step towards a good design. Furthermore, we hoped it would facilitate better methods of communication between the managers. Another approach would consist of a number of global function calls for any particular action, instead of logical messages through the appropriate manager's protocol.

13.3.3 OOPS

Object-Oriented Programming Systems (OOPS) are currently the best tool for a large software design. The concepts behind OOPS provide the best mechanisms for a system such as the TVS. Encapsulation allows the managers to keep their state private. Class aggregation lends itself perfectly with the DIS environment (the entity type enumeration, and basic vehicle dynamics models).

13.3.4 Soil Sample Objects

Because we have already decided upon an object-oriented design, it was obvious to encapsulate the soil sample objects. However, most experts in the field now agree that simple objects like this should not actually be encapsulated objects, but fully open public structures. In Object-Oriented Analysis and Design, Grady Booch writes:

If our abstraction represents a simple record of other objects and has no really interesting behavior that applies to the object as a whole, make it a structure. However, if our abstraction requires behavior more intense than just simple puts and gets of largely independent record items, then make it a class.

Therefore, we shouldn't encapsulate objects like the soil sample class.

13.3.5 Vehicle State and Vehicle Controls

The argument for this is similar to Soil Sample objects above. However, the vehicle state and vehicle controls classes are far more interesting and they encompass more behaviors than the soil sample class. The details of this can be easily found by looking in the header files or reading the TVS documentation.

13.3.6 Vehicle State and Controls Hierarchical Structure

It was an attempt at encapsulating some of the functionality of the vehicle type from the user. This is obviously the usual object-oriented approach to solving this problem. Furthermore, the vehicle hierarchy presents itself naturally to this methodology. In other words, it seems obvious to have a parent class containing the functionality common to all vehicle types (speed, acceleration, etc.).

13.3.7 FORMS library used for the GUI.

FORMS, by Mark Overmars, was designed specifically for quick and easy GUI development. It also works very well (as many of IST's programs use it). Rather than spend much time trying to develop our own user interface, we decided to implement FORMS. At first, it was a challenge to get FORMS and Performer working together, but it turned out well.

13.3.8 Command pipes were implemented

It turned out that the managers (especially the GUI and IG) needed to communicate far more than originally planned. In fact, it was formerly thought that no communication was to take place between them. This was a fallacy, however, since either manager (or both) could receive keyboard and mouse commands from the user. We want one place to handle these events, however. Therefore, we implemented command pipes, using the UNIX pipe() command (for more information, refer to the TVS documentation). The identities of the pipe are stored easily in shared memory. Each pipe requires only two integers, one to describe the read pipe and one to describe the write pipe. While this decision still seems like a good idea, it is unknown whether or not the pipe() function is completely reliable. Word was received from Bob Buckley that it may not be. In fact, the pipe() architecture was actually taken out of BwanaVision (BV). However, so far in all of our tests, this construct seems completely reliable.

13.3.9 Reuse of BwanaVision

At the beginning of this project, BV was not finished. We did not rely on BV totally as the IG. Instead, we created our own IG and merely implemented much of the code from BV. Some of this code includes the Entity Manager and Entity Container objects, Performer objects (fog, etc.), all DIS objects, and the Dynamic Terrain objects. We now view this as a good decision because BV is highly specific and consumes much of the systems resources.

13.3.10 The Entity Service Used

The only other choice was to use DIP (the DIS Interface Program) for reading the network. However, this is highly specialized for BV with in the loop synchronization for the IG process. Therefore, an interface to the Entity Service was created through the BV EntityManager object (as a sibling class of the NTSC_EntityManager). The obvious benefits for using Entity Service are:

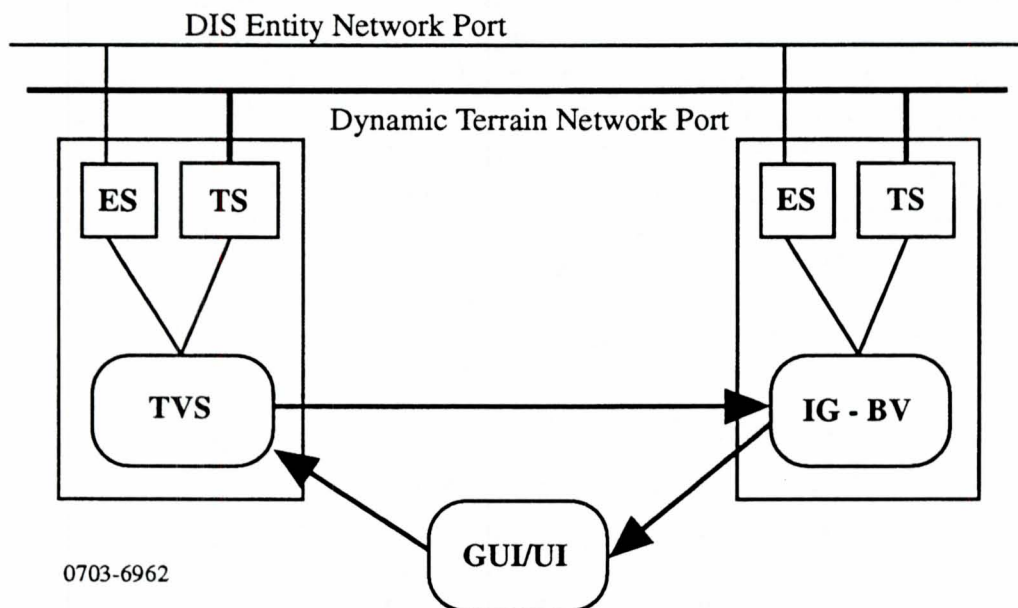
- created and maintained by IST personnel,
- able to service more than one program which allows multiple simulations to run on the same machine (unlike BV),
- and easier to update for new PDUs.

13.4 MVC Proposal

In the highest level, the Model is the application, the View is the Image Generator (IG), and the Controller is the User Interface (UI or GUI for Graphics User Interface). I propose to separate the View into another process and have it communicate with the application and Controller via shared memory constructs (whatever means are deemed best and most readily available; including AT&T shared memory, files, pipes, and Performer shared memory).

13.4.1 Simulation Overview

The overview of the simulation can be readily expressed by the figure below. It shows the Host/



IG/UI configuration, but with the actual components of our TVS defined. The IG will be BwanaVision, as described below. The GUI will also be displayed through BwanaVision, but the UI (possibly hardware interface controls) could reside elsewhere.

The host, which houses the TVS, also runs a copy of the Entity Service and Terrain Service programs. Thus, the TVS has connections to the network. Also, the host does not communicate all network traffic back to the IG for display, but gives the IG its on connections to the network ports. Of course, due to the incredible flexibility of Entity Service and Terrain Service, the IG can still

Figure 57. Simulation Overview

reside in the same machine as the TVS with no change in the protocol or system design. The host and IG are shown as separate processes on separate machines in the fig

ure, however, to better illustrate the power of this approach. Though the GUI is also shown as a separate process, it might not actually be detached. We allow the GUI to reside within the IG.

The UI is another story. The interface to the host should remain the same in either case. However, which process does the UI code reside in? The choices are: its own process, the host, the IG. If the GUI is actually in the IG, then it might be easier to just put all of the user interface code in the IG and this would simplify things. The problem with this thought is that part of the IG, the part which corresponds to the UI, would need to be modified for every new vehicle we implement. The host already needs to be modified. No matter how we design the system, the UI will need to be updated or modified whenever a new vehicle is implemented.

13.4.1.1 IG

For the new design, it was decided to revise BwanaVision (BV) and use it as the IG. BV would then need to be modified to communicate with Entity Service and Terrain Service, rather the DIP and an internal version of Terrain Service, respectively. The first has already been accomplished in version 0.6 of the current TVS. The latter change has been attempted in version 0.7, but the result was unsuccessful. The benefit of this decision was (stable) code reuse from BV. The DTMOBJ also needs revision. Furthermore, some of the unneeded functionality of BV may be left out, or not; this includes ownership objects without a host.

13.4.1.2 GUI

It was also decided not to use FORMS for the GUI, but switch to the new Performer supplied widgets. They are just as easy to use and provide faster update and input speeds. FORMS is commonly known to be a slow interface. We will lose the ability to visually generate the interfaces, but this is a small thing to lose in comparison to the new widget architecture. One gain is the use of possibly X based widgets. In other words, the interface will operate on any system which has Performer. This includes OpenGL, X, and any other future releases.

It is important to note that the GUI resides within BV. BV will do the input queuing and so forth, conveniently extracting any commands which are purely IG dependent (frame rate control, screen size, window location, etc.). Some commands, like keyboard presses (function keys) and mouse buttons and mouse location are entered directly through the IG window and the IG is the only process/object which has access to these events. Therefore, it seems natural to consolidate all input events, with a beneficial screening process, in the IG. It also seems likely to implement some sort of the GUI selection system through the inter-process control scheme (probably shared memory) between the TVS and BV. The GUI need only start up after connection with the host interface, which is the Tracked Vehicle Simulator.

The Booch class diagram for the new Tracked Vehicle Simulator can be found in Figure 58. The rounded rectangle shown in the figure corresponds to abstract classes. The definition for each class is also given below.

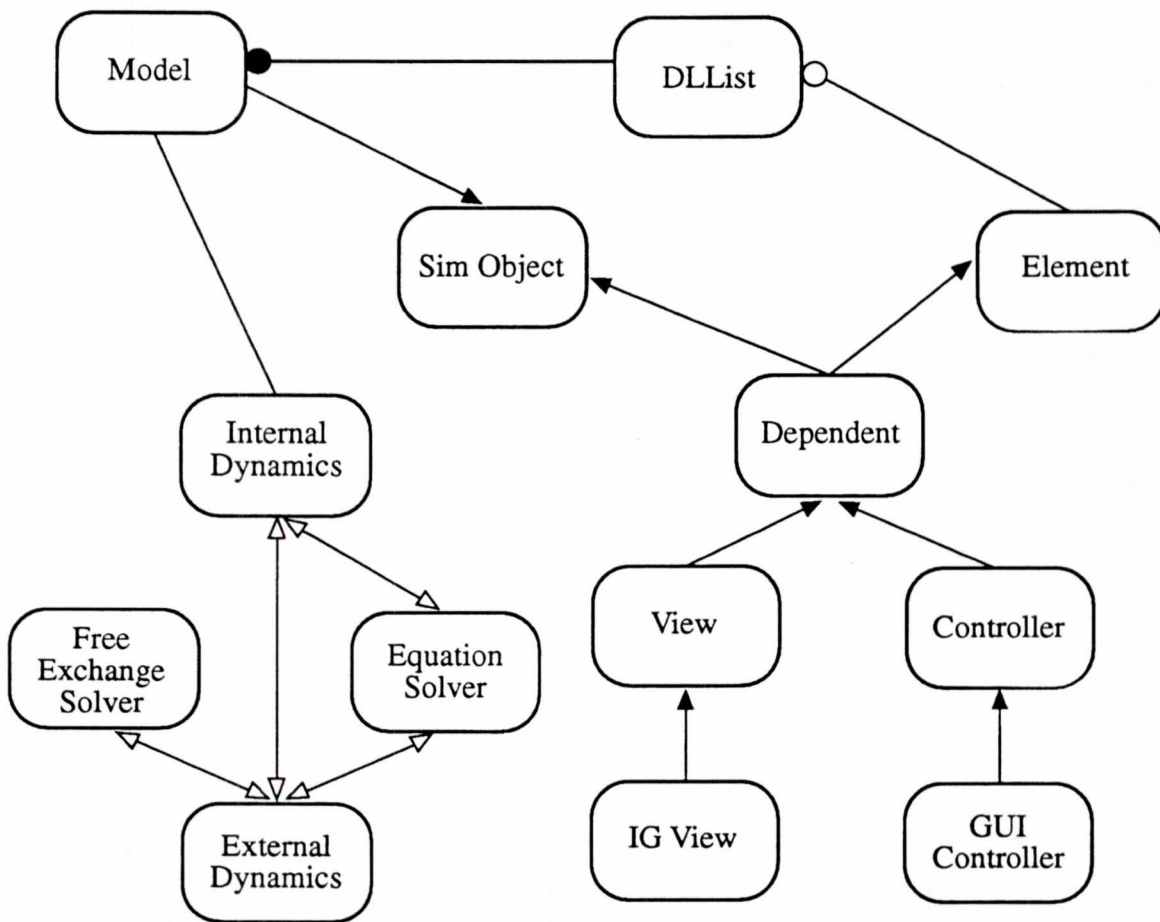
13.4.2 Diagram Description

13.4.2.1 Sim Object

This is the top-level simulation object. The two methods included here, to be polymorphed by any of its children are *printOn* and *sim*. The first provides a means to print out information to stdout for debugging or other useful purposes. The latter method allows for simulation control from an outside source. This is also known as “pinging” the simulation object to perform one loop of control, or to evaluate itself and its dependents in some way.

13.4.2.2 Dependent

This class encompasses the View and Controller classes, which are both dependents of the Model



0703-6897

class. Fundamental to the Model-View-Controller paradigm, the dependent hierarchy allows the Model class to have an unlimited number of views and controllers.

13.4.2.3 Model

This class controls the physical simulation of the system. Mobility, vehicle dynamics, collisions, and other aspects of a real-time simulation will be found in this object, or connected to this object through views and controllers to other model objects.

13.4.2.4 View

The View class represents a view of the model. This could mean a visual view, as for the IG, or just a view of state attributes of the physical model.

Figure 58. Class Diagram

13.4.2.5 Controller

The Controller class provides input and change variables into the model. This is better known as

the user interface. Controllers should be the only way the physical model can be controlled from an outside source such as a spaceball, mouse, or other device.

13.4.2.6 IG View

This is the view mechanism central to the Tracked Vehicle Simulator IG. It communicates via shared memory to the actual IG: BV.

13.4.2.7 GUI Controller

Similar to IG View, this object communicates with the interface controls.

13.4.2.8 Element and DLList

Some useful object-oriented tools for data structure manipulation have been developed. One such tool is a linked list utility. The linked list, DLList, recognizes its elements through the class Element. All objects which are to be used in linked lists need to inherit from this class. Two new functions, new_object() and del_object(), need to be polymorphed in the subclass for the list to work correctly.

13.5 Network Services

13.5.1 Entity Service

The entity service program provides a mechanism for obtaining all the standard DIS PDU information. This includes entity state, detonation, and fire PDUs. We have also designed and implemented a new PDU description, called the Attach PDU (also use for Detach). This PDU is described in other documentation. The state information for each entity should reflect the most current data on that entity. The position and orientation of the entities are dead reckoned by the Entity Service, but only if dead reckoning information is given in the PDU.

13.5.2 Terrain Service

The Terrain Service program provides a flexible means of getting up-to-date database coordinates and soil data. The terrain is stored in mathematical surface models and can be queried in various forms by the client application. For further information on the Terrain Service, please refer to the corresponding document.

13.6 Simulation Host

The simulation host is essentially the modified Tracked Vehicle Simulator (TVS). It will not be using any API other than the C++ code. The special API used in the system will be encountered in the IG host and the GUI.

The vehicle dynamics will be modeled in the simulation host. The exact location (not dead reckoned by Entity Service) of the vehicle will be known only to the vehicle dynamics module running in this host. In fact, it is this model which will be supplying the Entity Service with the remote entity approximation information. The TVS can support any of the dead reckoning algorithms of DIS.

Vehicle dynamics also includes modeling articulated parts. The bridge of the AVLB and HAB and the scooping arm of the Breacher, as well as other articulated parts of other vehicles, will be modeled by the TVS simulation host. However, once the bridge is detached (or some other entity on another vehicle) a special Dynamic Terrain Resource (DTR) program operating in another process will control that new entity. For instance, for the AVLB, bridge deployment was controlled by the Bridge DTR.

It is the simulation host's responsibility to communicate correctly to the entity and Terrain Service programs. It must be fully aware of the protocol used and any changes to this protocol during development and new releases. The simulation host must also communicate with the GUI over a special interface protocol which is fully detailed in another document.

13.7 Graphical User Interface

The graphical user interface (GUI) is the module of the simulator which is responsible for communication with the user. Input by keyboard or mouse will be received by the GUI, processed, and sent to the simulation host for controlling the vehicle or the simulation in some way. As of yet, this does not include other hardware input or control devices such as the spaceball, Polhemus head tracker, helmet, or FastTrak devices, which directly relate to the viewpoint rather than changes to the vehicle location or orientation.

The IG may also desire a graphical user interface rather than relying on text consoles as supported by the IG host. The GUI should then be expanded to encompass the needs of both the simulation host and the IG host.

The GUI can use two different and very powerful API libraries to accomplish its given tasks. These libraries are Performer and FORMS. Others are available but have been disregarded as not inappropriate.

13.7.1 FORMS

FORMS was the API of choice for the first iteration of the TVS design. It provides a very simple interface which is also powerful enough to handle most control capabilities for different vehicles. It also provides an excellent utility to help design the graphical interface which is displayed during program operation. This utility, the FORMS designer, outputs the necessary C code which is then compiled and linked into the rest of the simulation program. FORMS is written and updated by Mark Overmars and is public domain software for non-commercial purposes.

However, the main problem with FORMS is the fact that it can be incredibly slow to update. This

could be hazardous in getting timely inputs from the user. Furthermore, if the GUI is actually operating on the same machine as the IG, then it could also slow down the IG, especially if there is only one CPU on the system.

Due to the easy implementation of FORMS, it still may be considered as the primary choice of API for the TVS. The other option, Performer, is described next.

13.7.2 IRIS Performer

A new GUI API was released in version 1.2 of IRIS Performer. Performer is used in the IG and is discussed further in that respect below.

This API supplies some useful and very efficient functions for controlling the vehicle. As the name suggests, Performer provides far better performance than a FORMS interface and this could be decisive in the simulation.

However, performance may be the only issue which supports the Performer choice. FORMS is easier to implement and far easier to design an interface for. Performer apparently uses X window widgets to implement buttons and sliders and is comparably difficult to use.

13.8 Image Generator Host

The image generator (IG) host is not the same thing as the IG itself. The host merely communicate the entity, terrain, and viewpoint information to the IG. This piece of code is basically a peer application to the Tracked Vehicle Simulator and it needs to communicate to the entity and Terrain Service in the same ways.

It also needs to know the protocol of any IG it is capable of supporting. The two IGs currently supported are the modified ESIG 2000 and BwanaVision. These are detailed further in the next section.

13.9 Image Generator

The IG performs all the three dimensional rendering for the simulation and provides the user with the key feedback to how the simulation is running. This includes the vehicle which the user is operating. The information that the IG needs in order to continue is supplied by the IG host. The two IGs which we support are the ESIG 2000 and BwanaVision (BV).

Certain control input which is not supplied by the IG host can be entered directly to the IG by other means. A text terminal is connected directly to the Evans and Sutherland ESIG 2000, for instance. And a window terminal is supplied in the background by BV. More specific information on each IG follows.

13.9.1 ESIG 2000

Detailed documentation about the IG protocol used by the ESIG 2000 was supplied by Evans and Sutherland. Some of these functions have been implemented in the ESIG Host program. This code may be reused in the IG host process.

The ESIG 2000 is a separate hardware IG which must receive network updates through a special host. This is one purpose of the IG host, which may be generic enough to power other IGs as well as the ESIG 2000. However, these updates may slow down the usual 60Hz frame rate of the IG. Although the refresh rate is still around 60Hz, the entity and terrain updates only occur at about 15Hz or less. More information about the ESIG Host and the ESIG itself can be found in the respective documentation.

13.9.2 BwanaVision (BV)

BV is written using the IRIS Performer API for Silicon Graphics workstations. Version 1.x is the latest release of BV and its specific capabilities are covered by the documentation supplied by the original designer, Bob Buckley. Currently, BV does not easily conform to the IG host protocol control mechanism for controlling the BV IG.

Performance of dynamic terrain through Terrain Service and remote entity approximation through Entity Service visually by BV seem to be better than the ESIG 2000, despite the fact that the ESIG is a hardware IG and BV is a software IG operating on a Unix workstation. BV can easily operate at up to 30Hz on an Onyx-4 RE2 with 4 Raster Managers. Slower performance is achieved on lower end machines such as the Indigo and Indigo2, but this performance is comparable to the higher end Evans and Sutherland image generator. This performance differential is probably due to the IG Host - ESIG 2000 interface.

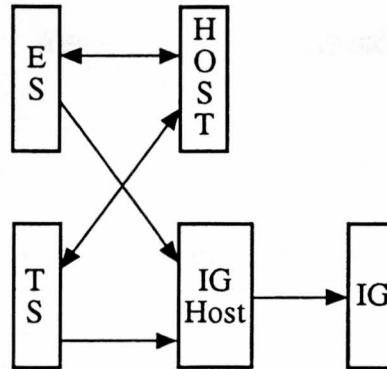
13.10 IG Host to IG to GUI Connectivity

We have decided upon a new design for the simulator, which differs from the classical design. This design does not yet include the graphical user interface. The GUI and how it connects to the design is the discussion for the rest of this paper. The design without the GUI is shown below. The detailed explanation of how this figure was created is given in the previous section, entitled *Host-IG Connectivity Design*. However, a brief review follows.

Figure 59. Current Design of Host-IG

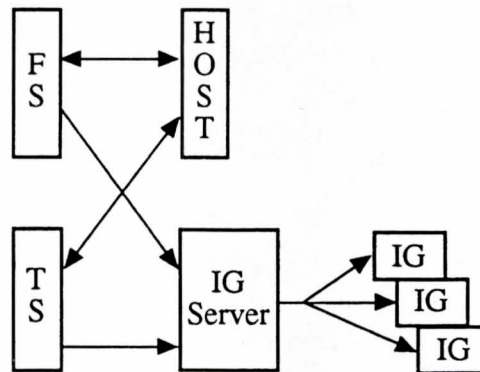
The Entity Service and Terrain Service processes both need to communicate with the host simulation. The simulation may also need to send information such as entities and terrain updates back to the network.

For proper visualization, the IG also needs access to the entity and terrain information. It was proposed and tentatively accepted that an IG host be devised which would receive entity and terrain information from the services and then route this information correctly to the attached IG, be it BwanaVision or the ESIG 2000. This would free up extraneous processing by the host simulation.



0703-6898

Furthermore, it coincides with our concept of entity and Terrain Service. These services are assumed to have the most up-to-date state information available to all peer applications. Thus, the IG (through the IG host) can get the viewpoint information about the host through Entity Service, instead of having a direct connection. There is a one way data passage from the IG host to the IG itself. This simplifies the IG protocol and provides an easy mechanism for IG substitution.



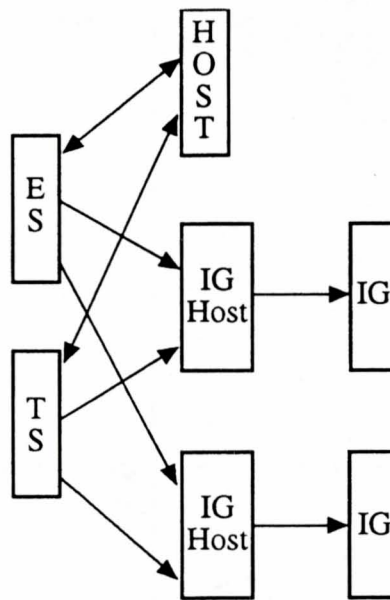
0703-6899

Figure 60. Host with an IG Server

The designs presented also introduces a new idea about the IG host. Instead of an IG host which communicates to a single IG, we can use an IG *server* which possesses the ability to communicate with several image generators. Figure 57 presents a diagrammatic view of this system.

However, this does present a problem. Different image generators which are controlled by the single IG server will be tied closely together. This could provide a serious bottleneck for faster IG systems. Furthermore, it is a fairly simple task to merely begin another IG host to control a new image generator. This IG host can then be the controller for another IG for the simulation host. The result of this conclusion is shown below. This figure represents the final suggestion for the Host-IG connectivity design. Of course, more than just two IG hosts and image generators may be

used.



0703-6900

Figure 61. Best Design with Host-IG

13.10.1 Overview of Alternate Designs for the Simulation System

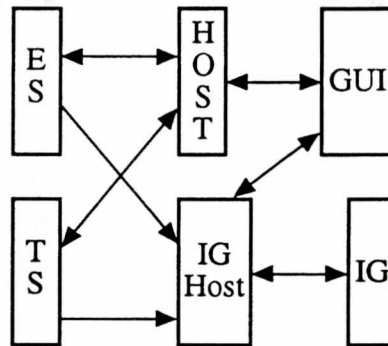
Several alternate designs were examined. Each of the suggested designs are given below. However, a discussion of each design may be necessary for correct interpretation. These descriptions precede the designs. Should the reader not desire a review of design alternatives, the final design is described in a later section.

The designs also contain a list of pros and cons describing the benefits and detriments of each design. While they should be self-explanatory, some may be further detailed in the following subsections of this report.

13.10.1.1 Design 1

The first assumption made is that the GUI will be displayed on an IG. Therefore, the first design presents the most obvious placement of the GUI. The problem with displaying the graphical controls on the IG is that the IG will now need to return input from the user back through the IG host and thus to the GUI and simulation host. If the GUI is separated from the simulation and IG host,

this could mean unnecessary complexity and latency problems.



0703-6901

Figure 62. Design 1

13.10.1.1.1 Pros

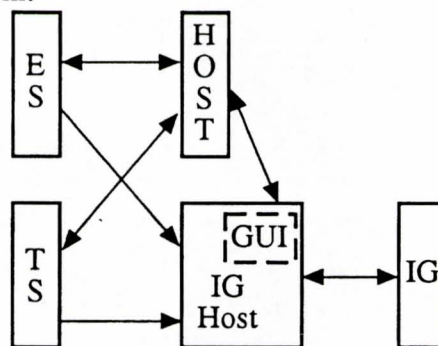
- Maintain GUI separately from the Host and IG
- Easy substitution for new input screens or devices

13.10.1.1.2 Cons

- GUI must communicate with both the Host and IG
- Added bidirectional pipe to the IG
- Input needs to propagate all the way to the host
- GUI must communicate with the IG for display

13.10.1.2 Design 2

In this design, we place the GUI controls closer to the display, thus eliminating a loop back step. However, now we also re-establish the link between the simulation host and the IG host. Furthermore, we tie the GUI to the IG framerate and vice versa, which could slow down performance on the IG, which is a major problem.



0703-6902

Figure 63. Design 2

13.10.1.2.1 Pros

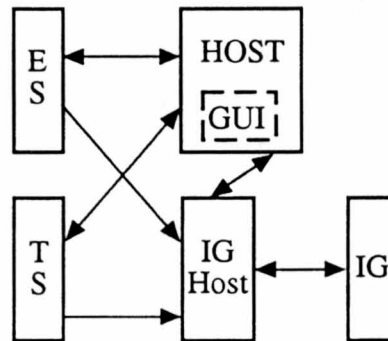
- GUI is coupled with its display
- Allows for a quicker GUI, if it is necessary

13.10.1.2.2 Cons

- Re-establishing the link between the Host and IG host
- The GUI does not require an update rate equal to the IG
- Added bidirectional pipe to the IG
- Changes to the GUI mean changes to the IG host

13.10.1.3 Design 3

The other option is to place the GUI in the simulation host instead of the IG host. This still establishes the link between them, but no longer ties the GUI to the IG. However, there is no real benefit of this design over that of Design 2 and they both have too many problems.



0703-6903

Figure 64. Design 3

13.10.1.3.1 Pros

- Controls can be more closely tied to the vehicle [dynamics]

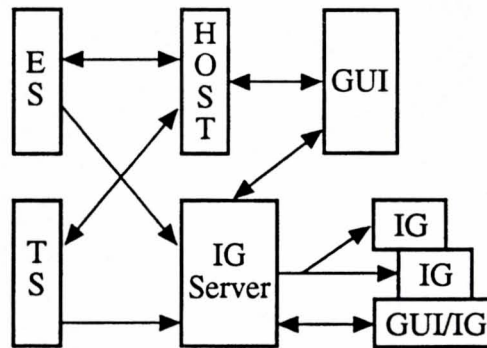
13.10.1.3.2 Cons

- Re-establishing the link between the Host and IG host
- Added dual pipe to the IG
- Changes to the GUI mean changes to the IG host
- GUI must communicate with the IG for display
- Input needs to propagate all the way to the host

13.10.1.4 Design 4

The problems associated with the two previous designs brings us back to the first design. We try to accommodate for the deficiencies in the first design with another design for the IG. Here, we present two types of IG. One which is just an IG, and one which displays IG information (entities,

terrain, etc.) and GUI information. However, most of the problems associated with Design 1 still exist.



0703-6904

Figure 65. Design 4

13.10.1.4.1 Pros

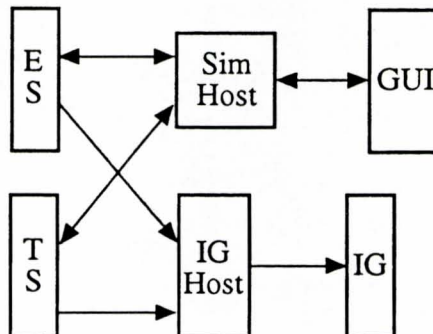
- Allows for multiple IGs, and thus only one which accepts/returns GUI input
- Offers support of the ESIG
- Maintain GUI separately from the Host and IG
- Easy substitution for new input screens or devices
- Dual pipe to the IG is only on the GUI display

13.10.1.4.2 Cons

- Input needs to propagate all the way to the host
- GUI must communicate with the IG for display
- GUI must communicate with both the Host and IG

13.10.1.5 Design 5

This design is the first one which separates the GUI from the IG. The GUI can now display its own information, and indeed it must do so in order to interact with the user. This idea will be prevalent in the next designs and in the final proposal. The problem with this design, however, is that only one IG can be used.



0703-6905

Figure 66. Design 5

13.10.1.5.1 Pros

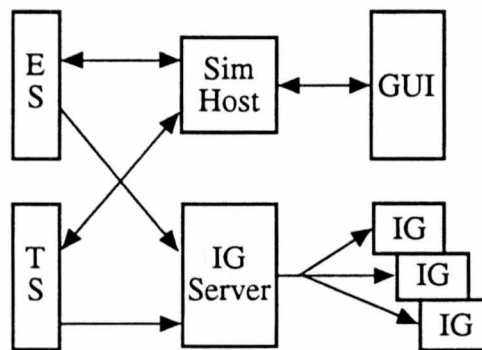
- Offers support of the ESIG
- Maintain GUI separately from the Host and IG
- Easy substitution for new input screens or devices
- GUI is coupled with its own display
- Allows for a quicker GUI, if it is necessary
- The GUI does not synchronize with the IG

13.10.1.5.2 Cons

- Allows only one IG

13.10.1.6 Design 6

Following the previous design, we can try to eliminating the outstanding problem of only one IG. Therefore, we modify the IG host by changing it into an IG server, able to communicate with several different image generators simultaneously. Unfortunately, this introduces a new problem of tying the different IGs together. They will all be updated sequentially by IG server.



0703-6906

Figure 67. Design 6

13.10.1.6.1 Pros

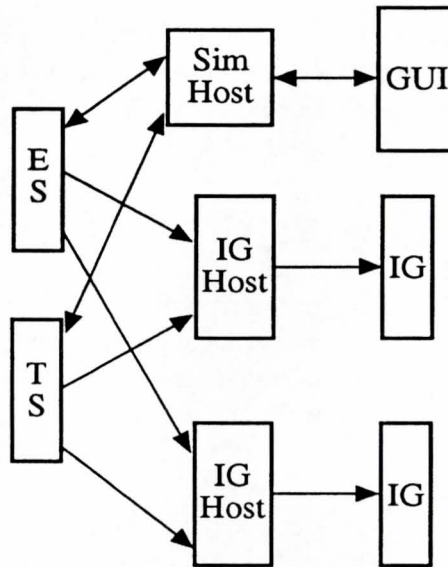
- Offers support of the ESIG
- Maintain GUI separately from the Host and IG
- Easy substitution for new input screens or devices
- GUI is coupled with its own display
- Allows for a quicker GUI, if it is necessary
- The GUI does not synchronize with the IG
- Allows for multiple IGs

13.10.1.6.2 Cons

- Ties the different IGs together

13.10.1.7 Design 7

This design represents the final proposal and includes the best ideas from the fifth and sixth designs. More than one IG can be used and they are not tied together. In fact, they will not even know that other IGs are running. Even the simulation host will not know about the IG(s)



0703-6907

Figure 68. Design 7

13.10.1.7.1 Pros

- Offers support of the ESIG
- Maintain GUI separately from the Host and IG
- Easy substitution for new input screens or devices
- GUI is coupled with its own display
- Allows for a quicker GUI, if it is necessary
- The GUI does not synchronize with the IG
- Allows for multiple IGs
- The IGs are not tied together

13.11 IG Host to IG Connectivity

13.11.1 Introduction

This section describes the possible variations in the Host-IG connectivity and overall design. It is important to note that this section does not discuss GUI placement. Our simulation will be using two external processes which are fundamental to our networking strategy. These processes are Entity Service (ES) and Terrain Service (TS).

13.11.2 Entity Service (ES)

ES provides a method of obtaining all entity information from the DIS network. It provides dead reckoning of all entities and encapsulated networking. ES also provides detonation, fire, and any other miscellaneous PDUs such as the new attached PDU developed in the VSL.

ES is a very lightweight process for a small number of entities. We have not yet fully tested the processing time required by ES, but for about 20 entities and some explosions, it operates easily at 40Hz. For a higher dead reckoning algorithm (the above was with algorithm number 1 in DIS standards) or many more entities this performance may decrease.

13.11.3 Terrain Service (TS)

The TS process likewise encapsulated the networking of PDUs. However, this process only handles the VSL's special terrain patches transmitted through unicast. The *patch* is actually a mathematical surface which can reduce the amount of data sent on the network. Terrain PDUs use a different port than the rest of the DIS network. A much more complete description of TS (and of ES) can be found in other documents.

Because of the power of the TS process, the application which uses it can request any resolution of terrain it desires. For instance, if a vehicle simulation needs to perform terrain following and a mobility model, it might require half meter resolution about the vehicle and not need any information elsewhere. However, the IG requires all information about the vehicle. Especially in our implementation, BwanaVision can perform continuous level of detail and reduce the resolution of the terrain the further it is from the viewpoint, thus increasing the performance of the IG. This is indeed a powerful feature of both the Terrain Service and of our IG, and it should be exploited to the greatest effectiveness.

13.11.4 Description of Alternative Designs of the Host-IG Interface

The designs of each of the suggested designs are given below. However, a discussion of each design may be necessary for correct interpretation. These descriptions precede the designs. Should the reader not desire a review of design alternatives, the final design is described in a later section.

The designs also contain a list of pros and cons describing the benefits and detriments of each design. While they should be self-explanatory, some may be further detailed in the following subsections of this report.

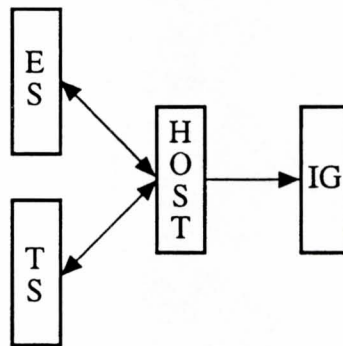
13.11.4.1 Design 1

Only the host applications connects to the Entity Service and Terrain Service processes. All network data which needs to reach the IG must go through the host application. The IG has no network capability of its own as far as the DIS network is concerned. It may, of course, communicate

to the host over the network, but that is outside the scope of this document.

Sending all network data through the host has its benefit: the host has the capability to manipulate the entity and terrain data before sending it to the IG. This may be considered a preprocessing step. The host would then have ultimate control of the IG, which is sometimes desirable. However, it might also duplicate most of the pre-processing done in either Entity Service or Terrain Service. This processing could be dead reckoning, culling, interpolation, or other things. Furthermore, resending all this data to the IG may require too much data transfer and processing time for the host which presumably already performs heavily CPU intensive calculations for a real-time dynamics simulation environment. It is unlikely that most machines could handle such a load for any sizable terrain database or large numbers of entities on the network. The biggest drawback to this design is the possible frame delay of viewing entity changes or terrain updates. Instead of these changes going directly to the IG when they happen, they must be processed by the host first.

Also, the IG may have the capability to perform continuous terrain level of detail swapping. This increases performance of the IG system by reducing the polygon count the further the data is from the viewpoint. This capability may not be possible if the host has first access to the data. The terrain updates would need to be sent to the IG at specific resolutions for proper continuous level of detail.



0703-6908

Figure 69. Design 1

13.11.4.1.1 Pros

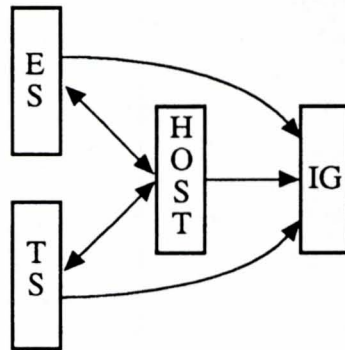
- Simple design
- Easy to reconfigure for other IGs
- Allows Host manipulation

13.11.4.1.2 Cons

- Requires too much data transfer
- Possible frame delay
- IG resolution of terrain limited to the Host
- No CLOD or the CLOD is expensive for the Host
- Extensive modification to BV is required

13.11.4.2 Design 2

The improvement this design presents over the previous one is the direct connection of the IG to the Entity and Terrain Service. The results of this connection should be obvious.



0703-6909

Figure 70. Design 2

13.11.4.2.1 Pros

- IG can get quick updates from the ES
- IG can poll TS with its own resolution
- IG protocol reduced
- CLOD possible and easy

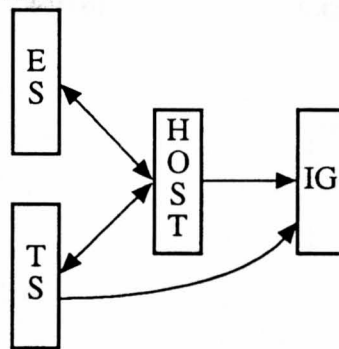
13.11.4.2.2 Cons

- No prior Host manipulation
- More difficult to integrate other IGs (like the ESIG)

13.11.4.3 Design 3

If it is determined that the processing of entities by the host is trivial with respect to computation time and that there is little or no recognizable frame delay in sending entity updates through the host to the IG, then Design 3 shows the revised version of Design 2. As you can see, the connection between the IG and Entity Service has been eliminated. However, the IG still gets terrain

updates directly from Terrain Service.



0703-6910

Figure 71. Design 3

13.11.4.3.1 Pros

- IG can poll TS with its own resolution
- IG protocol reduced
- Easier than Design 2 to reconfigure for other IGs
- Allows Host manipulation of entities
- CLOD possible and easy

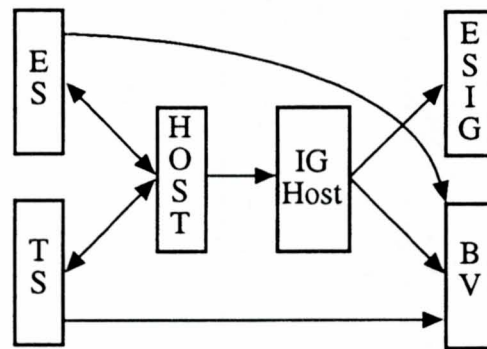
13.11.4.3.2 Cons

- Possible frame delay
- For large numbers of entities, this may also require too much data transfer
- More difficult to integrate other IGs (like the ESIG) with respect to TS

13.11.4.4 Design 4

This shows an extension of the previous design, with the two main IGs separated. The IG Host is a new concept similar in design to the ESIG Host. However, the IG Host can interface to more than just one type of IG. It can interface to either BV or the ESIG, with other future extensions possible. It is certain that the ESIG cannot connect to the Entity and Terrain Service on its own. It therefore needs the host interface for that purpose. BV, however, can communicate with the services because it is a software IG. The main problem with this design is the host must necessarily send all entity and terrain updates to the IG Host. It is not up to the host application to determine which IG it is powering or even the abilities of those IGs. That is solely the responsibility of the IG Host. This design will definitely complicate the system and it will not likely provide enough

benefits to outweigh the complication.



0703-6911

Figure 72. Design 4

13.11.4.4.1 Pros

- IG can poll TS with its own resolution for CLOD
- Same Host-IG protocol on the Host side for any IG

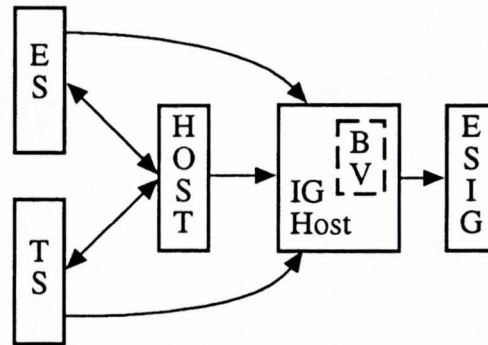
13.11.4.4.2 Cons

- Possible frame delay
- Complicates IG process
- Host still needs to update IG with entities and terrain, in case the IG does not support them itself
- IG has two connections to the ES and TS

13.11.4.5 Design 5

Never sending entity or terrain updates through the host application to the IG Host gives a much better method of the interface. The IG Host will now communicate to the Entity Service and Terrain Service itself to obtain all changes in the simulation. These updates will then be passed on to the IG. If the IG happens to be BwanaVision, then the update will be simplest. BV can then reside within the IG Host itself, and updates will occur quickest and with the least problems. However, the ESIG Host would then need to be specifically rewritten as part of the IG Host so that the IG

Host could communicate with the ESIG itself.



0703-6912

Figure 73. Design 5

13.11.4.5.1 Pros

- IG can poll TS with its own resolution
- Smart IG interface knows which IG is being used
- Entity and terrain data not sent to the IG
- Same IG protocol for any connected IG
- IG can get quick updates from the ES

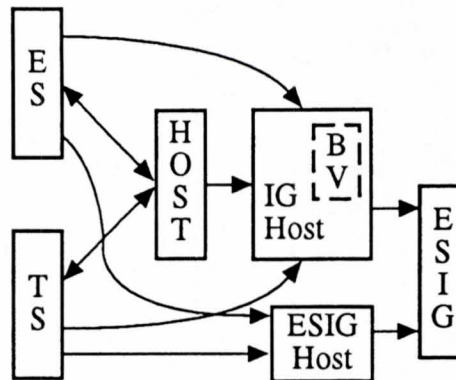
13.11.4.5.2 Cons

- Requires special handling for any IG, on a frame by frame basis

13.11.4.6 Design 6

The ESIG Host could also be a completely separate process running along side the IG Host. The IG Host would then drive the viewpoint and other important information updates of the ESIG, while the ESIG Host gives the ESIG entity and terrain information. BV would operate the same in this design as it did in Design 5. The obvious problem with this design is the fact that introducing another process like the ESIG Host requires another connection to the Entity and Terrain Service. This may not be a problem sometimes, but it usually will be a problem when all these processes run on the same single processor machine like our Indigos and Indigo2s. Obtaining entity information and especially terrain patch updates can become very expensive when the same CPU

needs to perform this function three times per frame.



0703-6913

Figure 74. Design 6

13.11.4.6.1 Pros

- IG can poll TS with its own resolution
- Smart IG interface knows which IG is being used
- Entity and terrain data not sent to the IG
- Encapsulates specific IG protocol in the ESIG Host
- IG can get quick updates from the ES

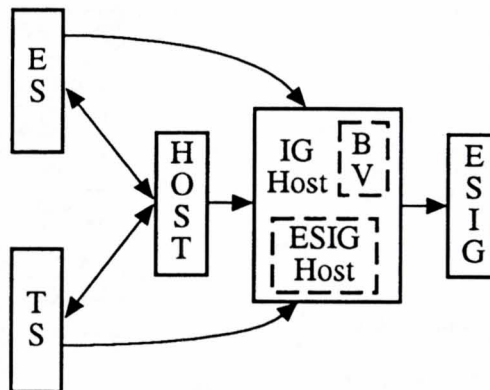
13.11.4.6.2 Cons

- Requires special handling for any IG, on a frame by frame basis
- An extra process needs to communicate with both ES and TS (ESIG Host)
- IG Host and ESIG Host may get confused on who's controlling the ESIG
- Possible conflicting commands
- Redundant processing of entity and terrain data

13.11.4.7 Design 7

In this design, the need for an external ESIG Host process is eliminated. Instead, the IGs supported by the IG Host all have their own host class running with the IG Host. This class knows precisely how to communicate with the IG. For the case of the ESIG, the ESIG Host within the IG Host knows to send terrain and entity updates across the ethernet connection to the image generator. For BwanaVision, the BV Host knows to keep all this information within local memory for

visual processing.



0703-6914

Figure 75. Design 7

13.11.4.7.1 Pros

- IG can poll TS with its own resolution
- Smart IG interface knows which IG is being used
- Entity and terrain data not sent to the IG
- Encapsulates specific IG protocol in the ESIG Host, but stays in the same process
- IG can get quick updates from the ES
- No redundant data for IG
- Same IG protocol for any connected IG
- No conflicting commands
- Allows manipulation of data at the IG Host level
- Easy to reconfigure for other IGs

13.11.4.7.2 Cons

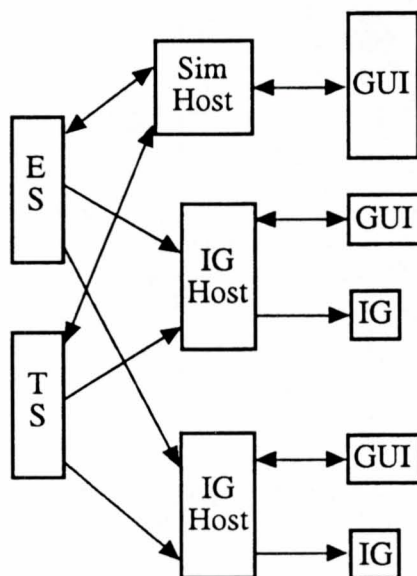
- Connection from Host to IG Host violates entity and Terrain Service' semantics

13.12 IG Design

13.12.1 Introduction

The simulation design suggested as a final proposal for the Tracked Vehicle Simulator (TVS) requires communication between the image generator (IG) host and the IG. This overall design is

shown below.



0703-6915

Figure 76. Current Simulation Design

13.12.2 Entities

The list of entities in the distributed simulation needs to be visualized by the IG. This list is updated and dead reckoned by the entity service. A coordinate conversion object will also be provided. The exact nature of this service program and the reasons for it are described elsewhere in another document.

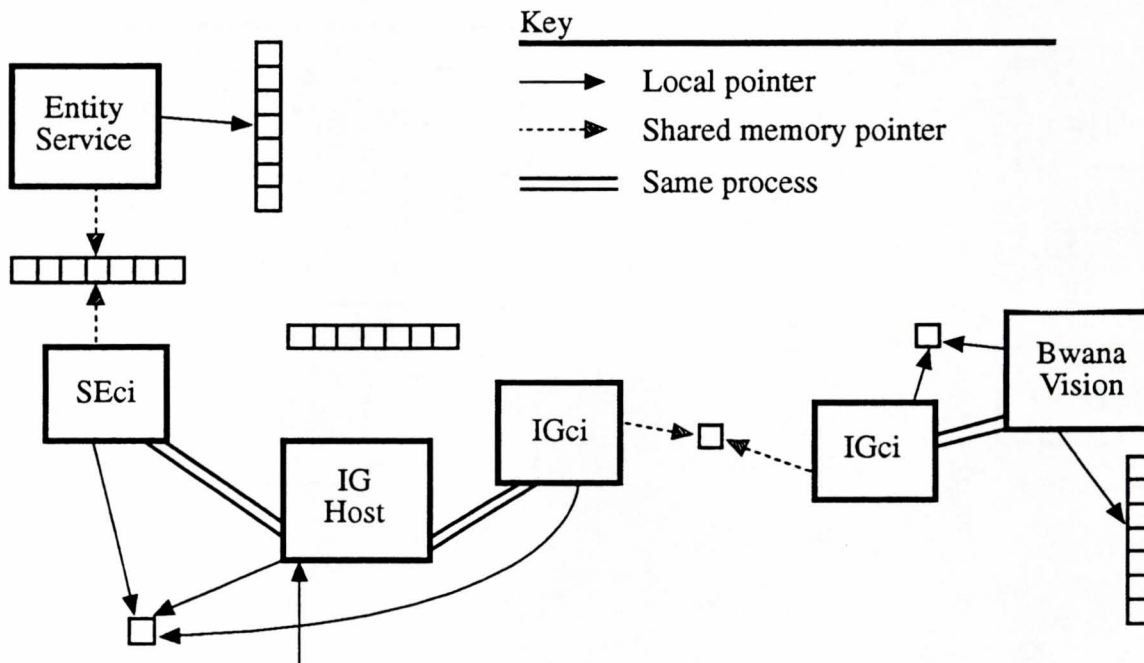
For the IG to visualize any of the entities in the simulation (including the one which is currently the host entity, if the IG host is not operating as a stealth) the entity list needs to be sent to the IG on a frame by frame basis. The dead reckoning occurs in the entity service, not in the IG, so the latest position and orientation must be obtained each frame. If this is not done, then “jerkiness” of the entity will occur on the visual display. There are basically only two approaches to take for the entity list: pass it through the IG host or directly to the IG.

13.12.2.1 Entities Through the IG Host

When the ESIG 2000 is the IG, the entities must be passed through the IG host. The ESIG does not yet have the capability to read entities from a DIS network and process them correctly. However, we are also concerned with the best solution for our Performer IG, BwanaVision (BV). Therefore, both types of entity processing should be considered.

We can group the entities together as a list or we can process them individually as they are passed through to the IG. Processing them individually provides a more *packet-oriented* approach, but it

is not nearly as efficient or clearly defined. Figure 77 shows this approach. As the entities are



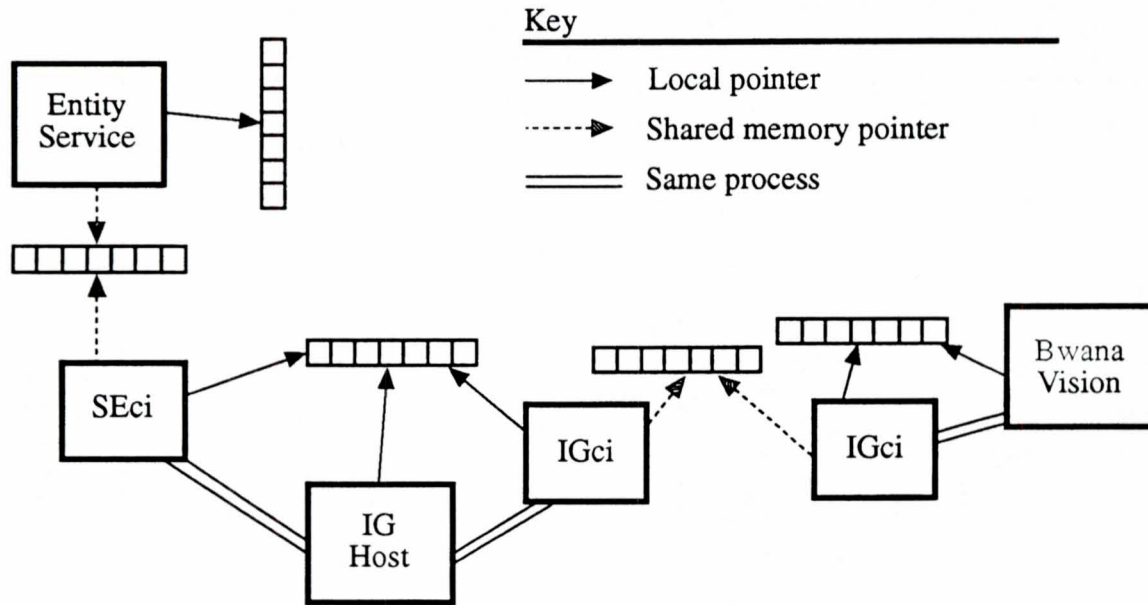
0703-6916

Figure 77. Tracking Entities Through the IG Host Individually

received, processed, and dead reckoned in the entity service, nothing is transferred to the IG host. The entities are only sent when requested through the Shared Environment communication interface (SEci). The entities are then copied from the entity service local buffer to the shared memory buffer between the SEci and the service. The SEci then copies these entities, one by one, as requested by the IG host, into a local buffer. The IG host then adds the entities into its local list as it gets each entity from the SEci. Once this process is done, the entities may then be downloaded to the IG. This will be done through the IG communication interface (IGci). Similar to the previous process, each entity is copied into a single local buffer which the IGci can read and summarily deposit into the shared memory or even network communication channel between the IGci and the IG (which will have a similar IGci on its end). On the IG end, the IGci will then copy the entity into a local buffer which the real IG (in our case, BV) will copy into its local storage buffer list.

A faster and more clearly defined method for passing the entity state information to the IG is shown below. Notice that the entities do not need to be individually copied from the communication interfaces to the application programs. The communication between the entity service and the SEci is the same for Figure 77 as it is for Figure 78. However, the list (in whatever format is decided to be used) is given to the SEci, which then fills in the entity state information directly, rather than doing it entity by entity. A reference to this list is also given to the IGci which

uses it to fill in another list in shared memory for final transfer to the IG's local memory buffer.



0703-6917

Figure 78. Tracking Entities Through the IG Host as Lists

A problem one may find with the entity state passing paradigm suggested above is a possible violation of the information hiding structure of object-oriented programming. If a class which represents the list of entities is implemented then the class structure is imposed upon both the communication interface classes and the application program. Of course, a class does not need to be implemented to represent the lists. A simple static array of entities could be used. This simple data structure would be more acceptable for any information hiding on the part of the application programs, but it is less efficient and more complicated. For the IG host to implement its own list class data structure, and communication with the SEci via a static array then another transfer of data would occur between the host and the communication interface. Furthermore, if even a simple array-based data structure is imposed then it may not be too detrimental in the way of information hiding to implement class-based lists. In fact, the choice would then narrow down to either the use of the paradigm suggested in Figure 77 or the above using classes.

13.12.2.2 Entities Directly to the IG

The quickest method of getting entity information to the IG is, of course, to have the IG directly communicate with the entity service itself. Unfortunately, this solution presents more problems than it solves. The first design implementing this idea is shown below. It shows the

direct connection of the IG to the entity service by individual entities.

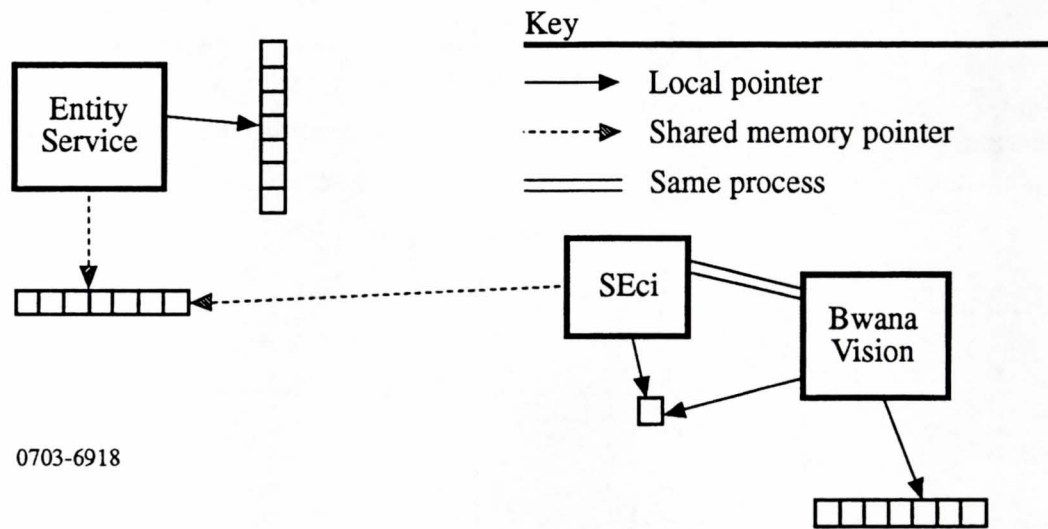


Figure 79. Tracking Entities Directly to the IG Individually

First, it reduces the complexity of the IG host. The IG host was design to reduce the complexity of the IG, not the other way around. BV currently is comprised of approximately 74,000 lines of C++ code. This huge amount can be daunting for the programmer just learning the system. We would like to see the IG segregated into logical, smaller parts. The idea of the IG host is an attempt at reducing the complexity of the IG. The actual division of work is considered in a later section.

Second, the entity state information which the IG receives may not correspond to the exact view point given by the IG host, especially if the IG represents a viewport for a specific vehicle (entity). A work around is for the IG host not to actually give the absolute view point, but merely an offset of the entity's base view point (position). However, this will not work for any IG but the one we write, BV. Highly specialized coding like this for multiple IGs is not a well defined structure for the IG host. It will lead to problems in the future when we update either the IG or the IG host.

We can help this process to be more efficient by introducing the list data structures. The figure above shows how the tracking of entities directly to the IG would look with lists instead of passing the entities individually from the SEci to the IG. While this may be the optimal solution, the problems presented previously should be considered.

13.12.3 Dynamic Terrain

Dynamic Terrain is an integral part of our current research. Any IG we implement must have the capability for visualizing changing terrain. Furthermore, although the ESIG 2000 can not implement continuous terrain level of detail yet, our image generator should implement CLOD. Unlike DIS entity states, the terrain updates can be very large and inefficient. Passing such a large

amount of data to the IG may not even be possible in certain situations.

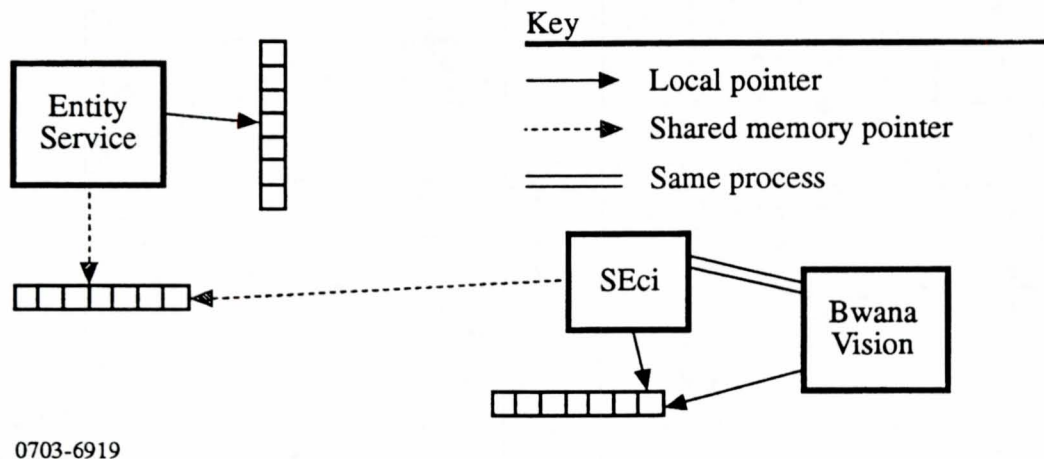


Figure 80. Tracking Entities Directly to the IG as a List

Consider a database which is 256 megabytes (MB). If the IG were just started it would somehow need to obtain all the database information to visualize whatever it needs. While we may consider the fact that this large transfer may only need to be done in the initial configuration of the IG, what happens when the IG is initialized sometime during the simulation rather than at the beginning? Any other process connected to the dynamic terrain service will be severely hurt by the transfer of data. Even if most of the terrain is culled by the IG host, there could still be a significant amount of data to transfer.

There are obviously advantages and disadvantages to passing the terrain data through the IG host for final visualization in the IG. We shall attempt to consider both methods briefly here and a better look at the segregation of responsibilities between the IG host and the IG will follow in the next section.

13.12.3.1 Dynamic Terrain Through the IG Host

Representing and updating continuous level of detail can be very complicated and may suggest a lot of data transfer. Unless there exists an efficient means of communicating the terrain updates to the IG, it may not even be possible to perform CLOD.

One method we may use to implement CLOD is shown below. The actual CLOD processing is performed in the IG Host process while it is imitated in the IG (more precisely, in BV, because the ESIG 2000 does not yet support continuous terrain level of detail). When the IG host determines that a row or column update needs to occur, it samples the terrain service itself and sends the updated row or column to the IG. There may be more than one row or column update. In fact, if the view point travels far enough, then the whole level of detail terrain patch may be updated, causing a possibly large transfer of data to the IG. This problem needed to be considered in our design.

Another problem one may readily notice with the figure below is the duplication of effort for

maintaining the CLOD. We can eliminate this by placing another direction of control from the IG to the IG Host, where the IG may request certain updates from the IG host to the terrain service process. However, this is a blatant violation of the fundamental design of the IG and may not be used. It would be impossible to suggest that every new IG implement this direct approach for dynamic terrain.

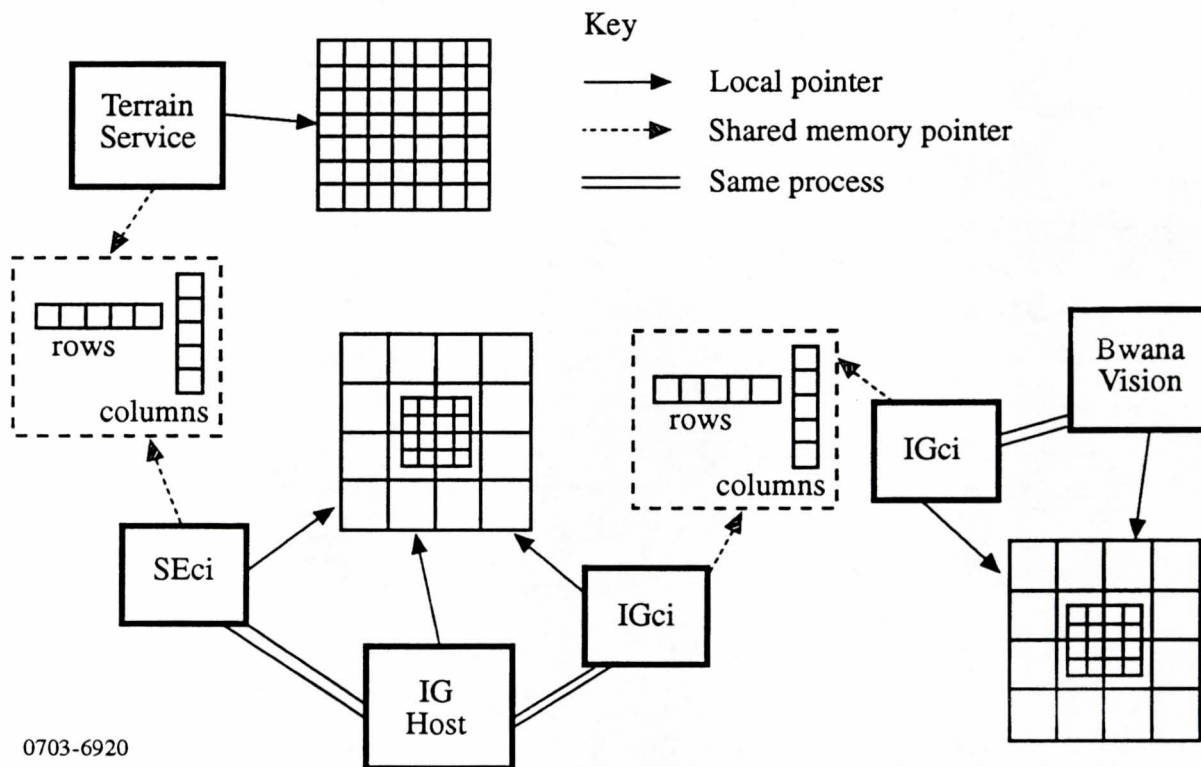


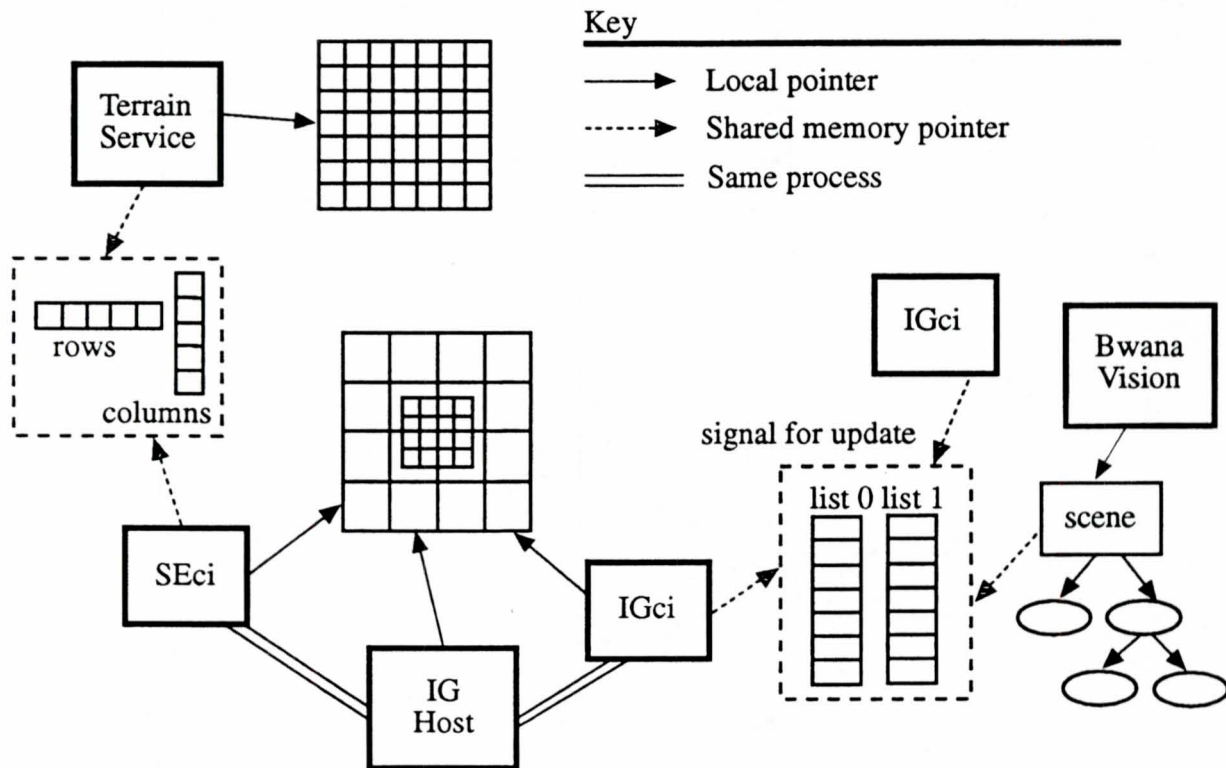
Figure 81. Tracking DT Through the IG Host by CLOD

Another possible design for passing dynamic terrain through the IG host to the IG is shown below. Though this design may seem attractive at the outset, it will probably not be acceptable for most IG implementations. Unless we know that the IG will be running in shared memory along side the IG host and that the IG will be using IRIS Performer as an application programmer interface (API), then we cannot use the design suggested above..

However, it may help us if we analyze the design a little further. Currently, version 1.0 of BwanaVision implements dynamic terrain in a similar manner. Another process manipulates the continuous terrain level of detail and communicates via the network to the dynamic terrain port. No terrain service is actually running. This process, the Dynamic Terrain Manager (DTM) Object of BV, communicates directly to the network to get terrain updates. In Figure 82 one can see the resemblance of the IG host to the DTM Object.

The IG Host has a certain amount of Performer processing to complete to update the Performer geometry trees in the shared memory segment between the IG host and the IG. One of the lists shown is the one to be currently displayed by the IG. The other is being manipulated by the process which calculates the CLOD. When the CLOD processing is done and the Performer list is

fully generated, a signal is sent to the IG process which then just swaps the two lists (by setting a flag in the root node; this is easily done by using the Performer pfSwitch node structure as the root node). This list is easily attached directly to the scene geometry of the IG, as shown in the figure.



0703-6921

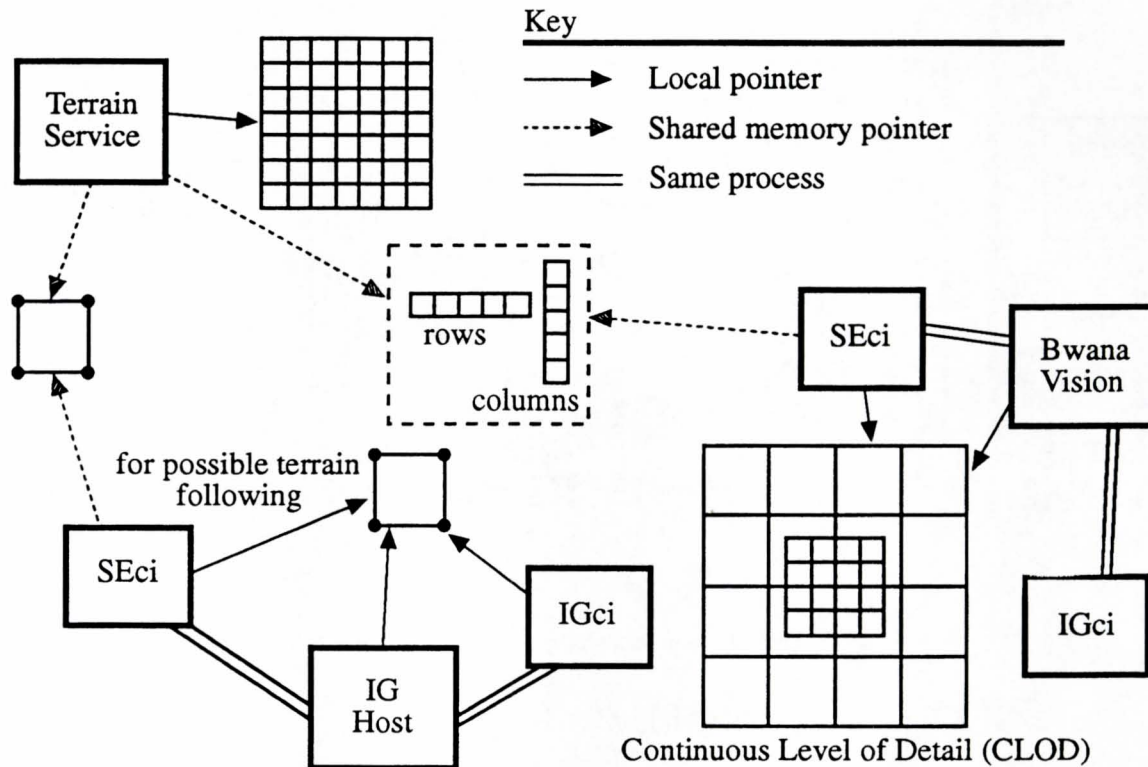
Figure 82. Tracking DT Through the IG Host by Performer Lists

However, if we intend to implement any other IG besides BV, then this design will be totally impossible. Moreover, it will seriously affect any type of dynamic terrain we attempt to emulate on other IGs. There will be two or more methods for implementing dynamic terrain for any IG and it will become unnecessarily complex to do so. The previous figure demonstrates a far better approach to the design of the IG and IG host.

13.12.3.2 Dynamic Terrain Directly to the IG

Instead of passing a huge amount of data from the IG host to the IG, we can design the IG such that it needs to get its information from the terrain service itself. Then, only terrain following needs to be performed in the IG host. However, this is not the solution we are looking for either. One of our goals is to reduce the complexity of the IG. The current version closely resembles the design shown below. However, all of the terrain processing is done in the IG, including the terrain following. Furthermore, no other process has access to the process which communicates with the dynamic terrain network (because only one process on a UNIX machine may bind to a single port). Thus, this does represent a significant improvement over the current version, but there is not

much.



0703-6922

Figure 83. Tracking DT Directly to the IG

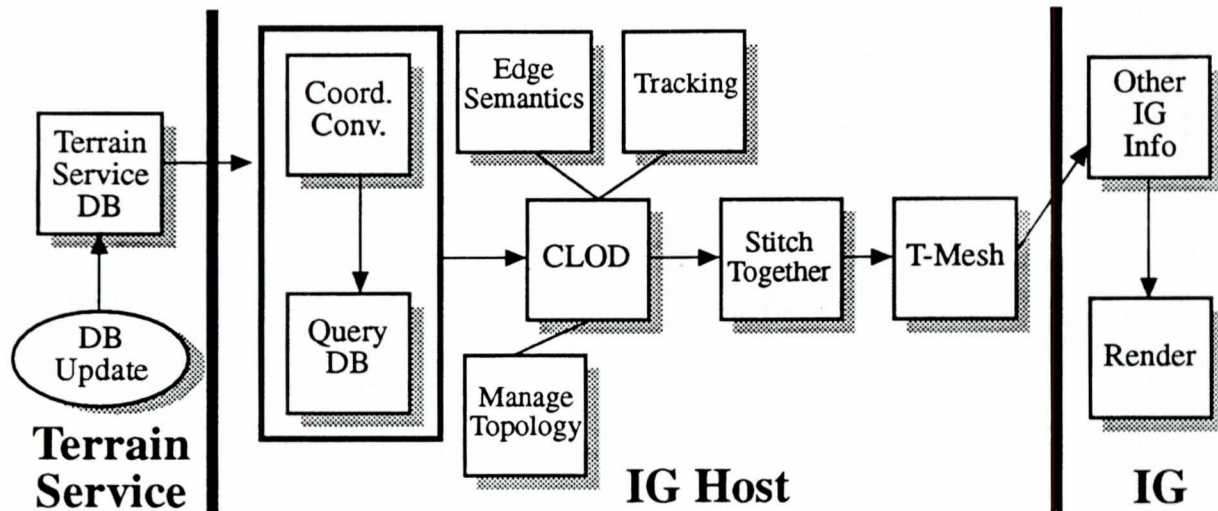
Although it is not shown in the figure above,, a DTM Object may still exist which will do the CLOD processing and generate Performer trees or lists in shared memory for the IG. Thus, a better approach than shown here may be a combination of both Figure 82 and Figure 83. However, we hope that a closer analysis in the following section will reveal a better approach to solving dynamic terrain and continuous terrain level of detail.

13.12.4 Segregation of Responsibilities

It is important to accurately define which process performs each duty for dynamic terrain and CLOD. It seems obvious that the querying for dynamic terrain to the terrain service process needs to be performed in the IG host process, rather than in the image generator. As has been suggested before, if we implement the ESIG 2000 IG, or any other IG besides BwanaVision, then the DT and CLOD functions need to be executed in the IG host regardless, so it makes sense to maintain those functions there. Rewriting and duplicating code in different areas is definitely a violation of one of the basic object-oriented paradigms for code reuse.

We can now define some of the elements of the CLOD pipeline. The figure below shows the individual components of the pipeline and the first suggestion for how they can be distributed between the processes. Before we can describe the segregation of functionality, the individual functions should be detailed more fully. A discussion of the figure follows the functional defini-

tions.



0703-6923

Figure 84. Dynamic Terrain Pipeline Version 1

13.12.4.1 CLOD Functional Detail

13.12.4.1.1 Terrain Service Database

This is merely the terrain service process which maintain the most up-to-date representation of the dynamic terrain database. How this database is stored may indeed be immaterial, but the reader can refer to the paper on the terrain service. For most applications, it will be represented as a mathematical surface, such as a Hermite surface patch or Non-Uniform Rational B-Spline (NURBS) surface patch.

13.12.4.1.2 Database Update

Corresponding to the terrain service, updates or changes to the database usually occur from outside sources. As far as the IG host and IG are concerned, all such changes will be made by outside sources, such as bulldozers, munitions, and many other sources.

13.12.4.1.3 Coordinate Conversion

Before a terrain query can reach its destination, the database coordinates may need to be converted into the correct reference frame. For example, if the IG can only represent the database in meters, but the database is stored in Universal Transverse Mercator (UTM) coordinates in the terrain service process, then the database query coordinates need to be transformed to the correct form.

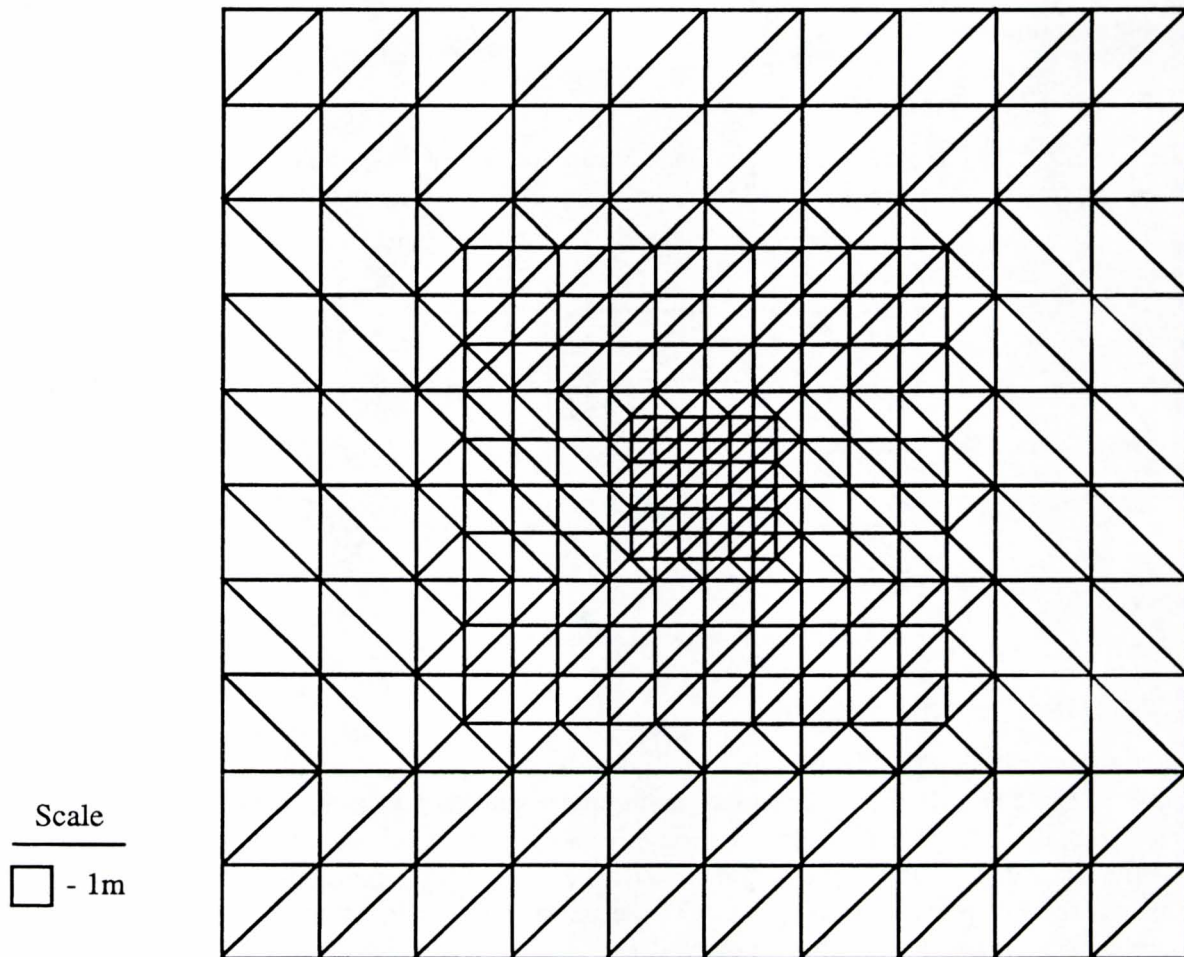
13.12.4.1.4 Query Database

Querying the database is performed in the IG Host process. This function uses the Coordinate

Conversion to put the database query in the proper coordinates before passing it off to the next stage in the CLOD pipeline.

13.12.4.1.5 Continuous Terrain Level of Detail

CLOD is process by which a certain portion of the database is shown in a higher detail than others. In our implementation of CLOD, the high detail areas surround the view point of the IG. As you get further away from the view point, the terrain has less and less detail, until at some point, the terrain is not even displayed. It may not be displayed at some point because it is outside the far clipping plane or outside the furthest level of detail. A complete example of CLOD is shown below.



0703-6924

Figure 85. Complete Example of CLOD

13.12.4.1.6 Edge Semantics

This function involves the querying and adjusting of edges (rows and columns), whenever they are needed.

13.12.4.1.7 Tracking

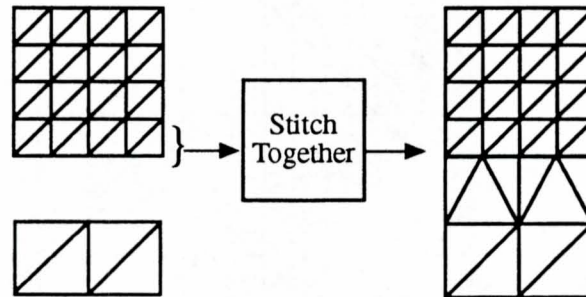
As mentioned above, the continuous level of detail follows the current view point of the IG, which is maintained by the IG host. Therefore, tracking of the view point is necessary for managing the CLOD for the terrain. Problems may arise if the view point changes too quickly for the CLOD processing.

13.12.4.1.8 Manage Topology

The terrain patches, or grids, are stored in an object class hierarchy. The grids are created and updated in an interior recursive fashion as detailed by Buckley in his Master's Thesis. The update of these grids (more specifically, the class structures) and how they react to each occurs in this stage of the pipeline.

13.12.4.1.9 Stitch Together

After the CLOD grids have been generated, or sectioned off, new polygons, or triangle meshes, need to be generated to "stitch together" the other patches. This is shown below.



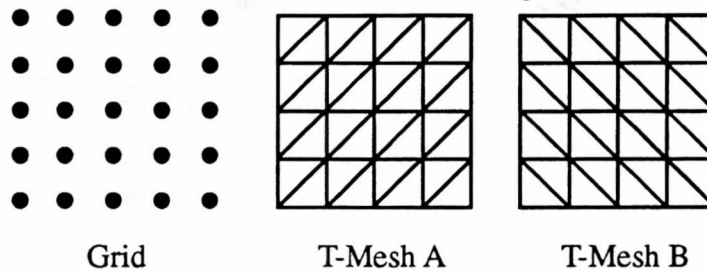
0703-6925

Figure 86. Stitching Together the CLOD Grids

13.12.4.1.10 Triangle Mesh

This functional component of the pipeline generates the triangle mesh out of the grid structure made by the CLOD algorithm. While it may seem obvious that a grid can be in fact interpreted as a triangle mesh, the triangle lines may need to go from one direction or another, depending on the IG. For example, consider the two triangle meshes shown below. Depending on various reasons, the IG may wish to display them differently. This may, in fact, be part of the IG host if the reasons are dependent on terrain specific information. If part of the terrain encompasses a ridge line, for

example, then it would be beneficial if the lines of the triangle mesh coincided with the ridge.



0703-6926

Figure 87. Creating Triangle Meshes from Grid

13.12.4.1.11 Other IG Information

This section part of the pipeline includes other terrain information such as color, texture coordinates, and infrared values.

13.12.4.1.12 Render

Render the polygonal information from the terrain pipeline to screen. This is the main function of the IG, of course.

13.12.4.2 Discussion of Version 1 of the DT Pipeline

Most of the functionality of the DT pipeline can be placed in one of the processes with reasonable assurance that they are in the correct place. *Correct* place refers to the most sound location, conceptually speaking, for that functionality to occur. All objects except for *CLOD*, *Stitch Together*, and *T-Mesh* should be in the correct places. Furthermore, the three questionable objects should be placed in either the IG host or the IG. No other location exists for their deployment. Of course, a spawned thread of one of those two processes may suffice and it would still remain an integral part of that process. For instance, the current Dynamic Terrain Manager Object of BV is only a spawned thread of the BV IG does not run independently.

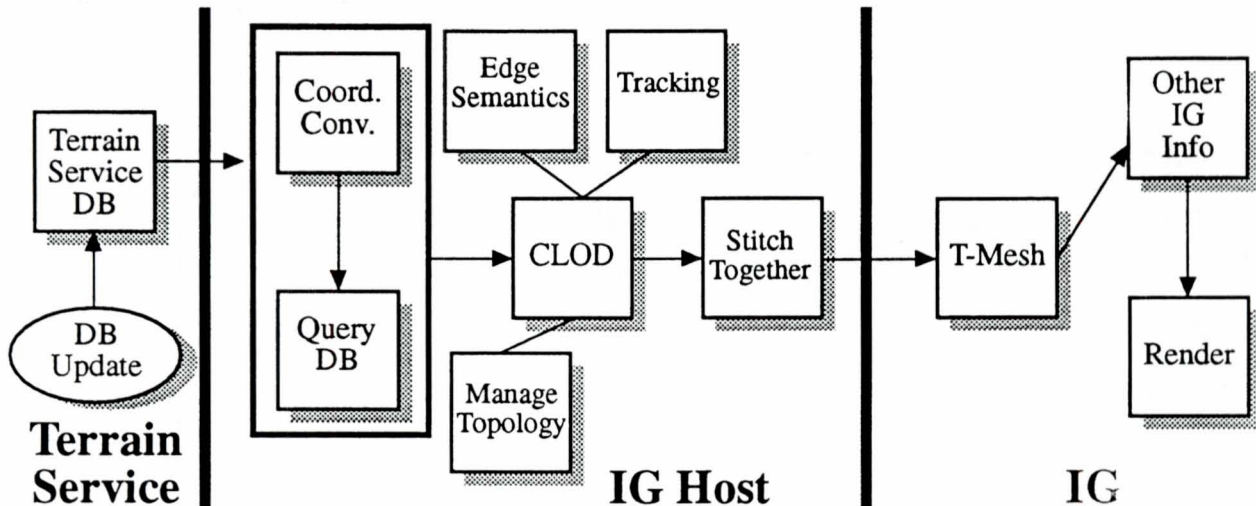
The big question we need to answer now is where do these three objects reside? As you can see in Figure 88, the first choice is to put them all in the IG host. Because of the nature of the pipeline, there are only four different groupings for these functional boxes. All three can go in one or the other process, or two of them can go in one or the other process (which also includes one of them being in one or the other process). However, we can cut this down by one choice by placing the *CLOD* object in only the IG host. This is possible and desirable because we wish to give more functionality to the IG host in order to reduce the complexity of our IG. Furthermore, it is more likely that new IGs will support various types of input, rather than perform CLOD on their own.

In keeping with the above statements, it seems likely we would want to place all of the *CLOD* functionality in the IG host, and give more time to the IG to perform its rendering. This is the main reason behind Version 1. Functionally, this would work as described by Figure 88 for our IG. Of course, for the current ESIG 2000, we could not even support continuous terrain level of

detail, so this version assumes that will always be the case. If it is not the case or may not be in the future, then Version 2 or 3 may be the better solution.

13.12.4.3 Discussion of Version 2 of the DT Pipeline

The next version we discuss places the *T-Mesh* function in the IG, rather than in the IG host. It seems logical that the conversion from grid points to a triangle mesh should occur in the process that best knows the precise description of the triangle mesh. Thus, there will be no duplication of effort to transfer from one type of triangle mesh to another. For the visual description of this version, please refer to the figure below.



0703-6927

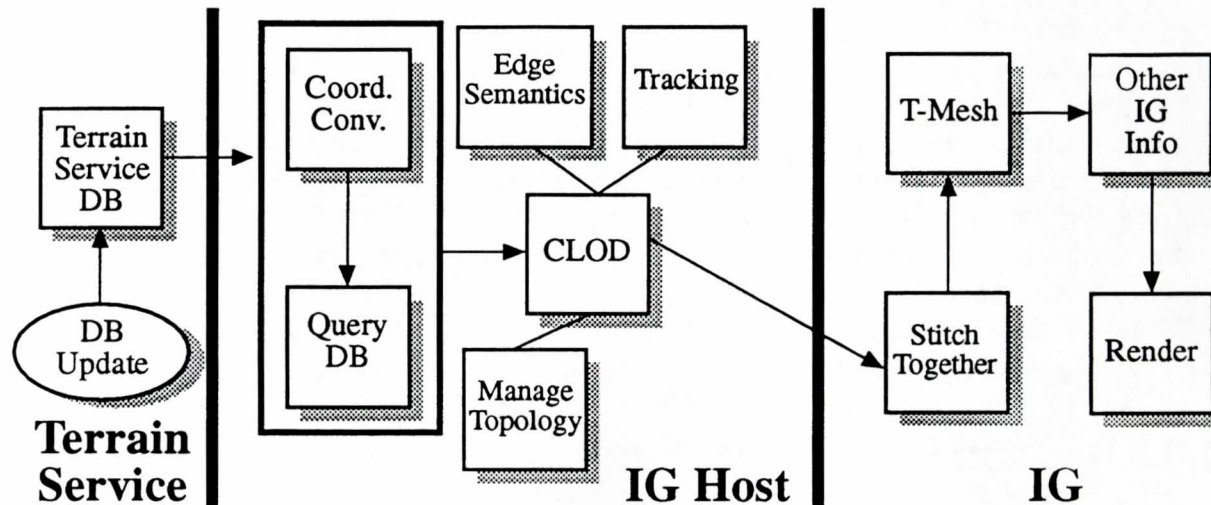
Figure 88. Dynamic Terrain Pipeline Version 2

If we were to implement the *T-Mesh* in the IG host, then either the IG host would implement its own protocol for the mesh so that it can send that information to the IG process, or it would implement the triangle mesh in the format which the IG uses, and pass that detailed information to the IG. The former implies a duplication of effort and the latter implies fundamental knowledge of the IG, as well as a very strong coupling of the IG host and IG. However, neither is very desirable, so it seems that Version 2 is a better solution than Version 1.

13.12.4.4 Discussion of Version 3 of the DT Pipeline

The *T-Mesh* object is not the only one capable of generating the terrain polygons. The varying spacing on the grid patterns of the CLOD make it difficult, if not impossible, to correctly triangle mesh them together. Thus, a new functionality is included which handles this task. The *Stitch Together* object generates polygons or smaller and independent triangle meshes which connect the different size grids together. For a visual description what is meant by this, please see Figure 86

and Figure 87. Please see the figure below for a visual description of Version 3.



0703-6928

Figure 89. Dynamic Terrain Pipeline Version 3

For the same reasons mentioned in the previous section on the discussion of Version 2, it may be the better solution to place the *Stitch Together* functionality in the IG, rather than in the IG host. This object does generate polygons or triangle meshes, so an inherent knowledge of the IG internal polygonal data structures would be necessary. Either that knowledge is necessary or duplica

tion of effort is required for the transference to and from mutually acceptable data structures. Of course, if the IG happens to internally implement the same data structure as the IG host, then our problem is solved, but this is so very unlikely as to make it not feasible.

13.12.5 Recommendations

For the tracking of entities from the entity service program, the solution represented by Figure 76 was selected. Although it may be a violation of information hiding to presuppose the data structures used by application programs. It is within the object-oriented programming paradigm to pass object classes between the communication interface and the application program. While, it may propagate bugs to the other programs, there should be no bugs in the fundamental data structure classes. At worst, it would require a recompile of both the communication interface and the application program when a bug in the list class is fixed. Furthermore, the application may still implement its own list structure by simply copying the data from the list class into its own structure. In this way, information hiding is not violated.

Also, because of the fact that for other image generators, most notably the ESIG 2000, we would need to track the entities through the IG host, it is undoubtedly the best solution to implement entities like this for any IG. If image generators become directly DIS-capable in the future, we

may change this, but for now IGs are not DIS-capable.

For tracking dynamic terrain patches from the terrain service program the solution as represented by the current simulation design was chosen. However, this is not precisely correct after reviewing "13.12.4 Segregation of Responsibilities". Rows and columns, per se, will not be sent to the IG. Instead, we should implement Version 3 of the three pipelines described in the previous section. Grid patches can be sent to the IG process, which will then use its own triangle meshing and stitching algorithms to create the displayable polygonal data structures. The implementation of dynamic terrain, and especially continuous terrain level of detail, has thus far proven to be resource consuming and CPU intensive. To this end, it is desirable to speed up the implementation of dynamic terrain. Duplicating the effort of either the *Stitching Together* or *T-Mesh* objects will be detrimental to the pipeline performance.

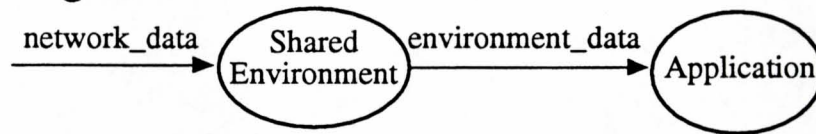
13.12.6 Data Flow Diagrams

13.12.6.1 Data Dictionary

network_data=	terrain_network_data, DIS_network_data
environment_data=	terrain_data, DIS_data
terrain_data=	raw_terrain_elevations, terrain_colors, soil_type
DIS_Data=	raw_entity_states, raw_detonations, raw_fires, raw_prototypical_PDU_data
displayable_data=	grids, view_point_data, bind_entities, update_entities, unbind_entities, misc_displayable_data, misc_control_data
view_point_data=	view_point_xyz, view_point_hpr
entity_data=	old_entities, clamped_new_entities, clamped_old_entities
misc_DIS_data=	detonations, fires, prototypical_PDU_data
misc_displayable_data=	explosions, other_special_effects
input_data=	misc_input_data, offset_select_data, ownship_select_data

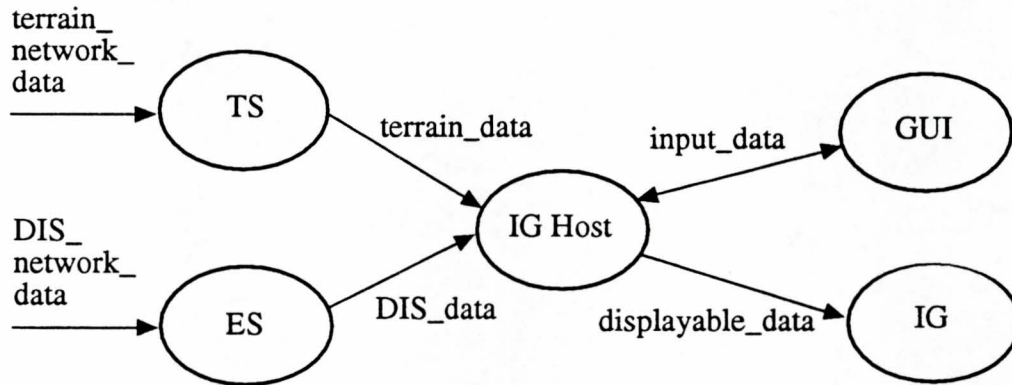
(Note: raw indicates an intermediate data format.)

13.12.6.2 Diagrams



0703-6929

Figure 90. Level 0



0703-6930

Figure 91. Level 1

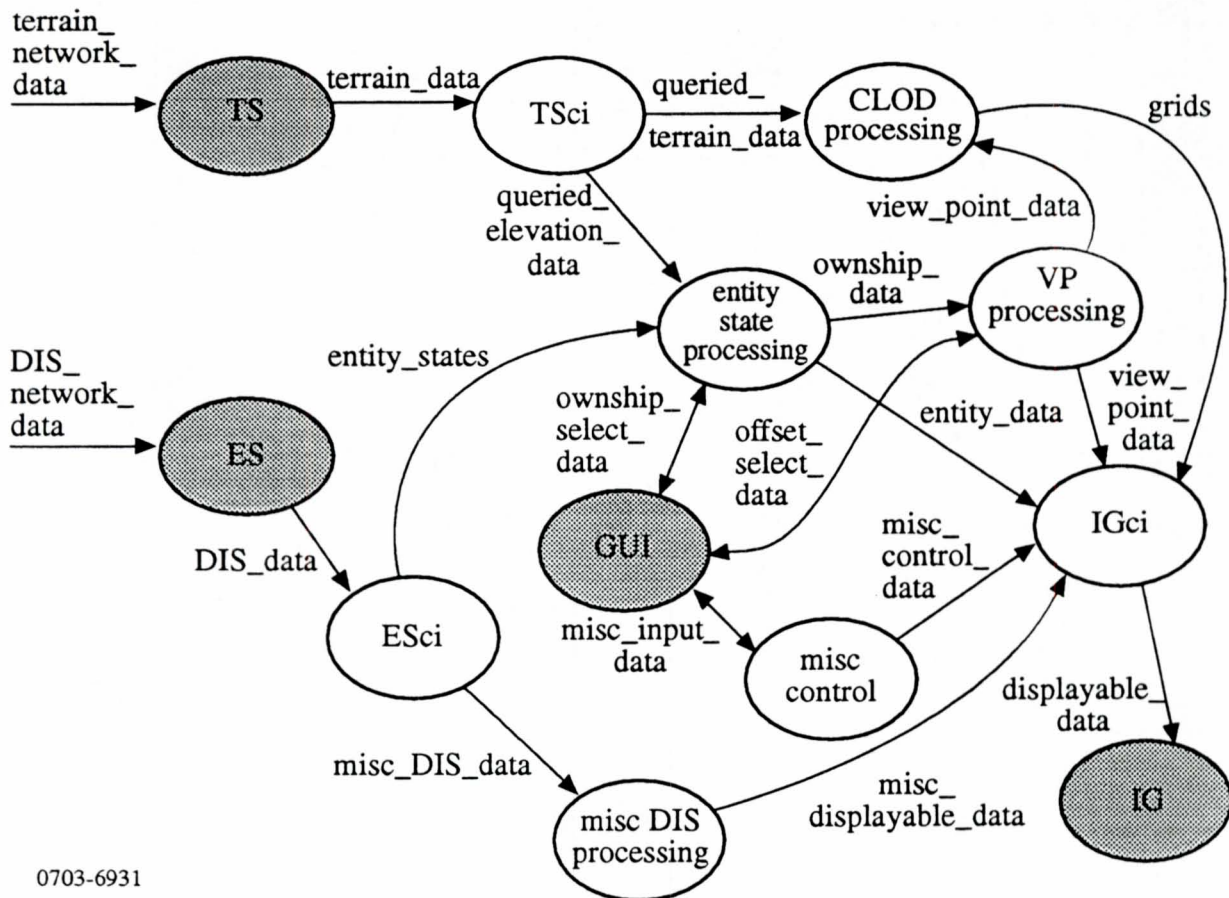
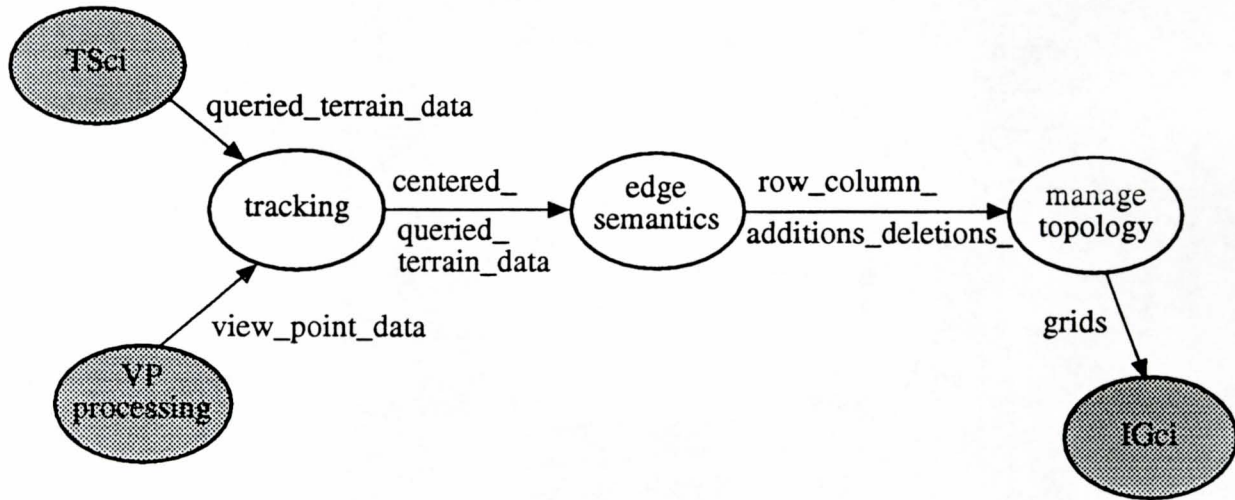
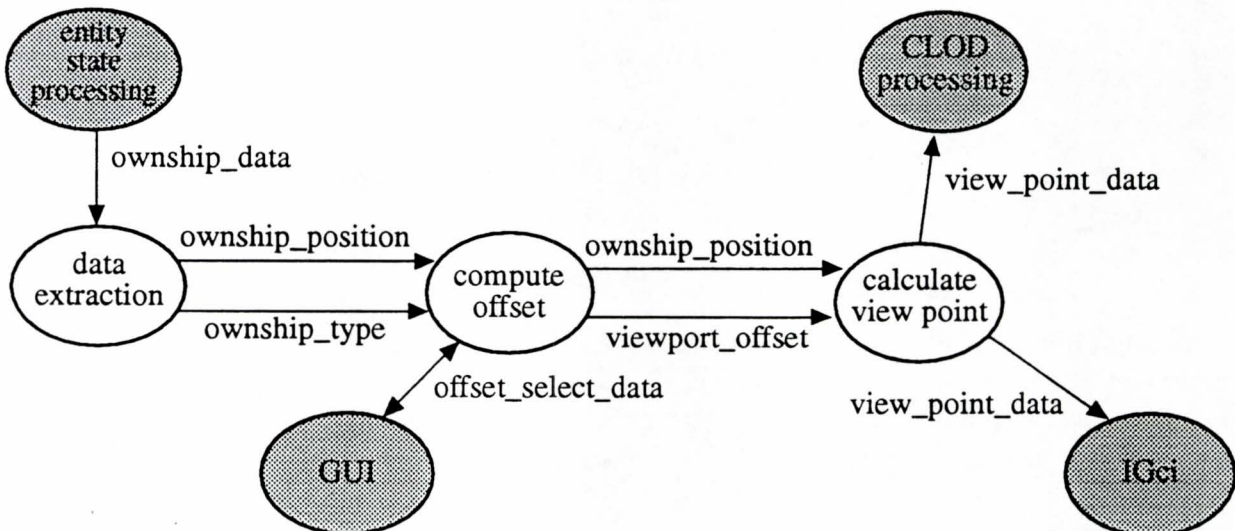


Figure 92.



0703-6932



0703-6933

Figure 93. (a)“CLOD processing” Level 3 (b)“VP processing” Level 3

13.13 IG Protocol

13.13.1 Introduction

The title of this section may be misleading for most people familiar with the simulation community's terminology. IG protocol usually defines the protocol from the simulation host to the image

generator. However, in this document we will define the protocol as something different.

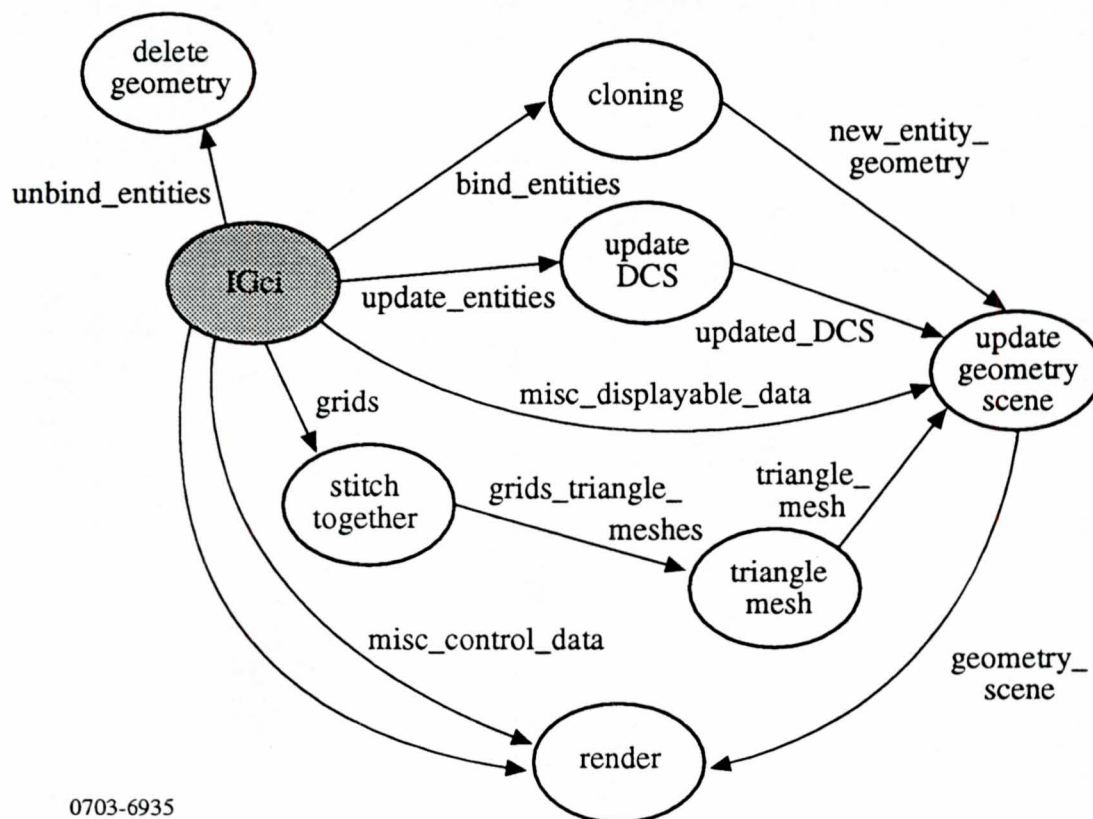
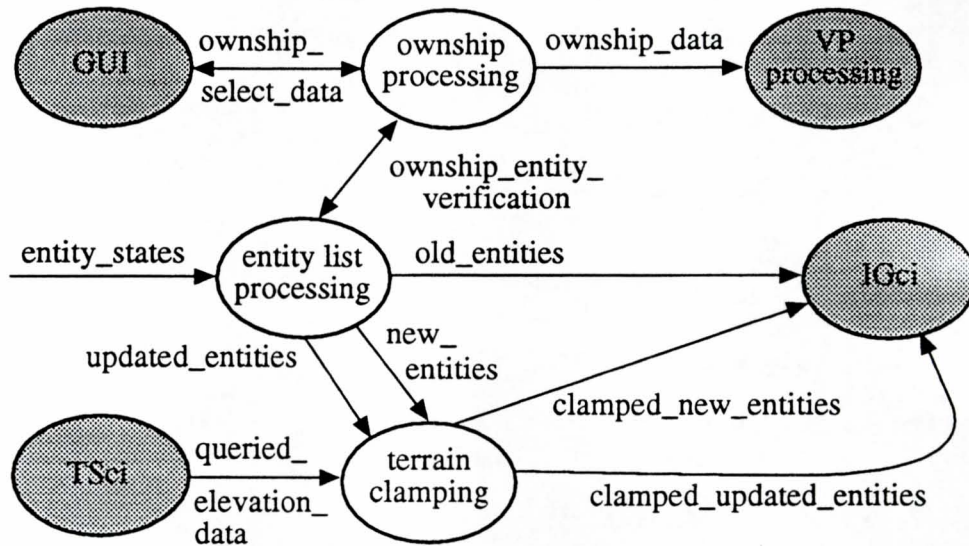


Figure 94. "entity state processing" Level 3

The design of our simulation system has been proposed as shown below. Note that communication between the simulation host and the IG has been removed as it is no longer a necessary connection. Instead, the IG host communicates with the network to obtain entity and terrain information which is then given to the host. The necessary host ownership information such as the view point and the orientation of the vehicle is obtained from the network entity state which is con-

trolled and dead reckoned by the Entity Service.



0703-6934

Figure 95. "IG" Level 3

Thus, there is no protocol directly between the simulation host and the image generator. Nor is there any protocol between the simulation host and the IG host. Of course, there is a network protocol for communication of entity state information (DIS), but that is another topic entirely and not discussed here.

The protocol between the IG host and the IG itself is completely dependent upon the designer and creator of the IG. For the case of the ESIG 2000, this would mean Evans and Sutherland. There are numerous IGs in use and most of them have different protocols for controlling them. In fact, there are even differences in communication between the ESIG 2000 and the ESIG 4000, which is a higher end IG. The protocol to BV is designed by the Visual Systems Lab. However, it does not have any support for outside control of entity state and terrain update information. It currently functions somewhat like our definition of an IG host. In essence, BV is its own IG host, as well as IG. Therefore, BV required some modification to fit this capability in its design.

13.13.2 What is IG Protocol?

The programming language of choice for the TVS is C++. This is a relatively powerful object-oriented language, despite the fact that it is not entirely object-based like Smalltalk. I mention this because it provides the concept behind this idea of IG protocol.

The IG host will be implemented with an IS-A object-oriented hierarchy with a child class for each IG. There will be a parent class called IG_Host which encompasses the IG protocol as its list of method operations. The actual IG host process will instantiate an IG host child of the type it needs. For example, if the user wishes to run BV for its simulation, then an IG host process must be configured for use with a BV_Host child class.

13.13.3 Classes with Suggested Protocol Methods

```
enum { NO_HOST, BV_HOST, ESIG2000_HOST } HostType;
```

13.13.3.1 IG_Host Class

```
class IG_Host
{
public:
    IG_Host( void );
    ~IG_Host( void );

    virtual void sendViewPoint( const float xyz[], const float
hpr[] ) {};
    virtual void sendCommand( const char cmd[] ) {};
    virtual void bindEntity( ulong id, const EntityType& type )
{};
    virtual void unbindEntity( ulong id ) {};
    virtual void sendSelect( ulong id, uchar select ) {};
    virtual void sendDetonation( const float xyz[], constEntityType& type ) {};
    virtual void sendFire( const float xyz, const EntityType&
type ) {};
    virtual void sendStorm( uchar stormLevel, uchar rainLevel,
uchar lightningLevel ) {};
    virtual void enableFog( uchar enPatchy, uchar height, uchar
thickness ) {};
    virtual void disableFog( void ) {};
    virtual void sendDT( const ulong *posts, const float *val-
ues, ulong count ) {};
    virtual void startAnim( uchar type, ulong id, const float
xyz[] ) {};
    virtual void stopAnim( ulong id ) {};
    virtual HostType IS-A( void ) { return NO_HOST; };

protected:
    Communication_Class *comm;

private:
};
```

13.13.3.2 BV_Host Class

```
class BV_Host : public IG_Host
{
public:
    BV_Host( void );
    ~BV_Host( void );

    // other functions declared here
    HostType IS-A( void ) { return BV_HOST; };

private:
```



```
};
```

13.13.3.3 ESIG2000_Host Class

```
class ESIG2000_Host : public IG_Host
{
    public:
        ESIG2000_Host( void );
        ~ESIG2000_Host( void );

        // other functions declared here
        HostType IS-A( void ) { return ESIG2000_HOST; };

    private:
};
```

13.13.3.4 Suggested Protocol Description

13.13.3.4.1 void sendViewPoint(const float xyz[], const float hpr[])

This command sets the viewpoint of the the IG to the world position specified by *xyz* and the orientation specified by *hpr*. Both variables are arrays of three floating point numbers, with the order given directly by the variable name. Thus, for the position, the x component is first, then the y component, then z.

13.13.3.4.2 void sendCommand(const char cmd[])

Send a text command to the IG. Use this function when you know the format of the command and there is no other call which performs the same function. Scanning and evaluating the ASCII command can be a relatively slow process as many comparisons are necessary. However, this provides an easy method to establish new features of the IG and implement special, IG specific commands.

13.13.3.4.3 void bindEntity(ulong id, const EntityType& type)

This function creates a new entity of the specific DIS type, which includes the *kind*, *domain*, *country*, *category*, *subcategory*, *specific*, and *extra* data fields. The entity is numbered as an unsigned long integer from 1 to 50,000. The IG host will then translate these entities into the correct numbering scheme for the corresponding visual IG. For example, the ESIG has entities numbered only from 3 to 255. Furthermore, some models, such as the AVLB, need a separate model for its articulated parts; in the case of the AVLB, this means the bridge. Therefore, another entity needs to be created strictly for the IG, which the host knows nothing about. In the IG host, these new models will be numbered from 50,001 to their maximum number able to be stored in an unsigned long integer. These numbers need to be further mapped for different IGs, depending on their numbering scheme. Again, for the ESIG, these models will be numbered from 255 down until there are no more models left.

13.13.3.4.4 void unbindEntity(ulong id)

Release the entity identified by *id* from the bound list in the IG host and the IG. This is the corresponding function to `bindEntity()` and it will usually be called when the entity times out.

13.13.3.4.5 void sendSelect(ulong id, uchar select)

Some IGs will support different appearances for the models. In effect, these are usually completely different models, but they are loaded together with a flag set to determine which one to display. One appearance may have a damaged or camouflaged look while another may be in perfectly good condition. Therefore, these models may be specifically identified by the *select* variable in the function call.

13.13.3.4.6 void sendDetonation(const float xyz[], constEntityType& type)

This function tells the IG host that a detonation has occurred and that the resulting explosion should be displayed. The world coordinate position of the detonation is given as *xyz*. The type of munition used to create the explosion is given in *type*.

13.13.3.4.7 void sendFire(const float xyz, const EntityType& type)

A weapon has been fired and any special effects which the IG possesses which could visualize the burst should be displayed. The world coordinate position of the detonation is given as *xyz*. The type of munition used is given in *type*.

13.13.3.4.8 void sendStorm(uchar stormLevel, uchar rainLevel, uchar lightningLevel)

This function is provided for compatibility with the ESIG command list. If the IG has the capability to provide visual thunderstorm effects, then they will be exploited. BV, for instance, can not do these special effects yet.

13.13.3.4.9 void enableFog(uchar enPatchy, uchar height, uchar thickness)

Enables the fog effects. When the fog is turned on by this function, the IG also needs to know if "patchy" fog should be enabled, and to what level of "patchiness". Patchy refers to varying densities in the ground fog. Patchy fog should be defined in the database, if any exists. The flag *enPatchy* will be 0 for no patchy fog, and will contain a value greater than 0 to turn on patchy fog. The actual value will determine the level of patchiness. The *height* variable gives the ground height of the fog in world coordinate units. The thickness also defines a parameterized range describing the fog. This value will be interpreted by the IG host and redefined for each specific IG. The range is 1 for very thin fog and 255 for the thickest.

13.13.3.4.10 void disableFog(void)

Turn off all fog. This also turns off patchy fog, if it has been enabled.

13.13.3.4.11 void sendDT(const ulong *posts, const float *values, ulong count)

This particular IG host command is directly related to the ESIG 2000 dynamic terrain modification utility provided by Evans & Sutherland. *sendDT()* sends a terrain update patch to the IG host, which then sends the terrain update to the IG. *posts* contains an array of enumerated posts in the terrain patch. The new values which are to be updated in the actual geometry of the database are stored in *values*. And *count* signifies how many posts are to be updated by this call. This function is not recommended. Terrain data updates are normally quite large. The IG should be capable of receiving its own updates from the network.

13.13.3.4.12 void startAnim(uchar type, ulong id, const float xyz[])

Other IG special effects animations besides explosion may exist. In particular, smoke and fire may be implemented as part of the IG rather than in a dynamic simulation. Therefore, this function allows for those animations. To keep from limiting the scope of the IG and enumerating every possible animation of every IG and accounting for future additions, the type of animation supported will be enumerated as different types. The values for this enumeration will be given in another document, as it is not important here. This function tells the IG to begin the animation and continue until a *stopAnim()* call is issued. The *type* of the animation is given on the parameter list. The animations should be numbered by the host and given a specific *id*. It is possible to given the same *id* to more than one animation. The effects of this action are further explained below in *stopAnim()*. The world position coordinates of the center of the animation are also given to the function in *xyz*.

13.13.3.4.13 void stopAnim(ulong id)

Ends the displaying of the particular animation identified by *id*. If animations have the same *id*, then this serves as a special grouping and all the animations who have the same *id* will be turned off at the same time. This makes things easier if many animations occur at once and have the exact same duration.

13.13.3.4.14 HostType IS-A(void)

This function must be implemented by the child subclass. It returns an enumerated value describing the type of IG host instantiated. For the values of the enumeration, see the preceding section.

13.13.4 The IG Host as a Stealth

The only communication between the IG Host and the simulation host are the network entity state PDUs. Technically, the simulation host will never really know if there is an IG Host (and thus an IG) which is providing the visual feedback for the human user. The IG Host process is initiated and maintained completely independent of the simulation. Whether there is one, two, more, or none at all makes no difference.

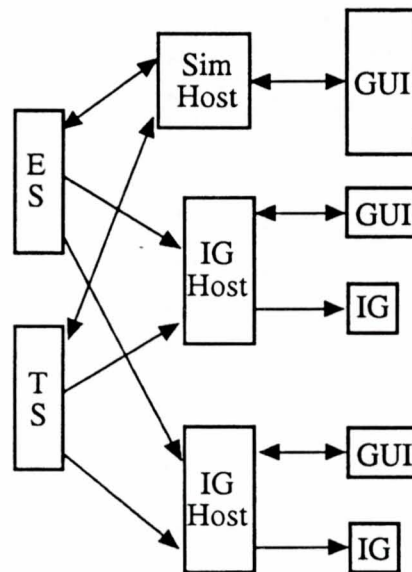
This indifference of the simulation host to the IG introduces an interesting capability of the simulation design. The IG Host can now comfortably and easily work as a stealth, as well as a view-port for the vehicle. A stealth operates as a window to the world. Everything that happens in the

world can be visualized by the stealth, which is controlled directly by a user. It is not a simulation host, so no entity can be generated by it nor can any occurrence in the simulation affect the stealth (such as fires and detonations).

13.13.5 Controlling the IG Host

Remember that the IG Host is a separate process from the simulation host and does not communicate to it directly. Therefore, the IG Host cannot receive control input information from either the simulation host or the graphical user interface. Furthermore, the IG Host does not have a bidirectional connection to the IG. No control input information can be obtained from that source either.

However, we do wish to provide the user with control input ability to the IG Host even after it has been initiated and configured (through a configuration file or command line parameters). Currently, a graphical user interface to the IG Host does not seem possible or even feasible. A text window, command line input, interface is much more likely. This interface will allow the user to run in stealth mode and possibly attach to different entities on the network. If we did wish to implement a GUI for the IG Host, the figure below may represent the right approach to this option.



0703-6936

Figure 96. IG Host with a GUI

13.13.6 Viewpoint Offsets

The most important aspect of the simulation as far as the IG Host and IG are concerned is the viewpoint. It describes, in a minimum of six floating point numbers, the location and direction of the viewer. This may be the forward view of the vehicle simulation, a stealth view, or even a window viewport of the vehicle itself.

The position of the vehicle itself is obtained from the entity state PDU received across the net-

work. Thus, the simulation host is no longer obligated to supply any viewing information about the vehicle. In fact, in most circumstances, the viewpoint *offset* needs to consider the polygonal geometry of the visual model of the vehicle itself. This offset is then added to the position and the correct viewpoint location is obtained. Also, viewports define certain offsets from the vehicle position and these must be located without the simulation host's help as well.

The problem we have now is to determine from where the offset information is to be obtained. The geometry will be loaded by the image generator itself, so the IG Host will not know ahead of time what is the corresponding offset information for the host vehicle. Furthermore, it is unreasonable to assume that the IG Host knows anything about the physical characteristics of the vehicle so that it may estimate the offsets.

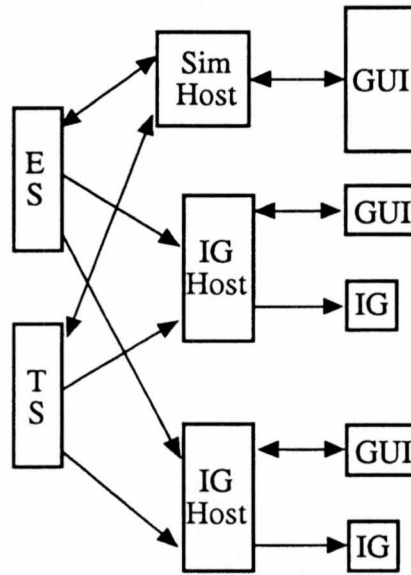
The only solution for this dilemma is to develop a table of offsets for each model on each IG. Thus, we would have a table for BwanaVision and one for the ESIG2000. It will be relatively easy to generate this table for BV as the vehicle models (in MultiGen Flight format) are readily available. Unfortunately, the ESIG2000's models are stored on the hardware IG itself and cannot be easily obtained. Therefore, that table will need to be generated model by model, viewport by viewport, by trial and error until it is sufficiently completed, at least for the vehicles we are considering simulating (Breachers, bulldozer, AVLB).

13.14 GUI Design

13.14.1 Introduction

The simulation design suggested as a final proposal for the Tracked Vehicle Simulator (TVS) requires communication between the simulation host, which is executing the vehicle dynamics and other physically modeled characteristics, and the graphical user interface (GUI). This design is shown below. Similar interfaces may exist for every image generator (IG) host which is running. The communication between these processes should remain conceptually the same as the communication between the GUI and the simulation host. Note that the communication lines between the simulation host and the GUI and between the IG hosts and the GUI are bidirectional. The protocol between the two processes needs to reflect this configuration. The reasons for the dual pipe should be obvious, but they will be explained further in the following sections. The simulation host and IG host will be referred to as simply the *host* so that it is not implied that only the

simulation host or IG host may communicate with the GUI.



0703-6937

Figure 97. Current Simulation Design

13.14.2 GUI as a Separate Process

A principle question of design is whether the GUI should reside in a separate process, in a spawned thread of the host, or as part of the host itself. It is obvious from the current design that we do not intend the GUI to be an extension in the host process. The benefits of having the interface in a separate process (rather than merely a spawned thread) are numerous, but the major drawback is equally large.

The simulation host dynamics and equation solvers need to run at time steps which are different from what the graphical user interface needs. In fact, the interface with the user can be very slow. The equations will need a very short time step, calculating integration equations at possibly 100 times a second. Therefore, tying the GUI to the simulation host would be a major drawback. Indeed, even calling a GUI update function for every frame of the dynamic simulation is preposterous for real-time performance.

Furthermore, the IG host may have a large responsibility to quickly update the IG. Sending terrain update information and entity information to the IG may require much time. If the IG host also had to update the GUI with its display function and user input processing, this may unnecessarily slow down the IG.

An answer to this problem is to put the GUI in a separate process and have it scheduled at a different time step. The GUI can now be event controlled or simulated at a pace slower than the host. However, the two processes still need to communicate via some method and it is this functionality which provides the greatest difficulty. The question of whether the communication between these processes is through shared memory or ethernet packets is immaterial as the communication

classes can be developed to eliminate this distinction. The figure below represents a conceptual view of the GUI and host communication channel. A detailed description of this

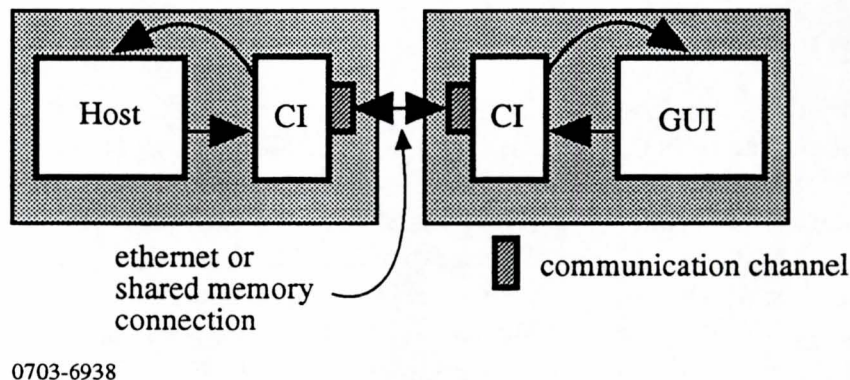


Figure 98. GUI to Host Connection

figure and its components can be found in section “Communication Description” on page 209.

The GUI represents the control panel for a specific vehicle or for the IG host (which may be considered as a *vehicle* here for simplification). In particular, it represents the vehicle being modeled by the host simulation. The information corresponding to the vehicle must be passed between the GUI and the simulation host. It presents a problem when we try to decide if all the vehicle information should be passed from the simulation or if just the vehicle type should be sent and the rest of the information (number of gears, type of transmission, etc.) can be read from a configuration file on disk. Either way, we still need to design a protocol which allows generic communication from the GUI to the host simulation. If this protocol is not generic, then we will have something like an AVLB GUI, a bulldozer GUI, and many others. Furthermore, the host simulation will need to be modified for use with every GUI we design. And if the interface changes for some reason (updated with new data from the original vehicle specifications, for instance), then both the host simulation and the graphical user interface need to be modified to reflect the changes.

If the GUI is localized in the host simulation, then all these changes will also be localized. Furthermore, the protocol will be reduced to inheritance relationships with polymorphed methods for communication. The correct GUI child class will be instantiated with the corresponding type of vehicle implemented in the host simulation process. However, this does not have the benefits of the multi-threaded environment described above.

Therefore, it was decided to adopt the multi-threaded environment and keep the GUI and the simulation in separate processes. The inheritance relationships discussed in the previous paragraph can be implemented even in separate threads. The following section reveals the details of this design.

13.14.3 Proposed Design

Control information must pass from the GUI to the host. Also, error indicator information must

then be sent to the interface from the host. The need for this bidirectional communication has already been established, but the specifics of how it is accomplished has not.

13.14.3.1 GUI State

The graphical user interface maintains variables reflecting the current state of the control systems. The collection of these variables can be called the GUI state. This concept is similar to the entity state in DIS or the concept of state in object oriented programming (having instance variables). The precise definition of GUI state will not be decided yet, especially since it depends upon the vehicle type or the IG host control panel. Of course, both the GUI and the simulation host know which vehicle is being simulated, so they must agree upon the state structure for each vehicle. It is unlikely that the state will be the same for any two vehicles, but this does not present a problem. It is sufficient to say that there exists a state for the GUI input controls.

The communication between the two process will occur through the new communication channel class defined elsewhere. The benefit of this class is that it encapsulates the form of transmission. If the channel is operating over the ethernet network or even through shared memory on the same machine, it does not matter; this is hidden from the process (in this case that would mean the GUI and the host). Therefore, we need a structure which is impervious to this type of treatment. Obviously, a continuous, frame by frame, stream of information could be detrimental over the network. Also, it may not even be possible to get a very small threshold of time steps for this continuous stream. Moreover, it is quite obvious that we will be using continuous control functions in our simulation and even for controlling the IG host. Some examples of this are steering and throttle control for the simulation host and head mounted display and virtual data gloves for the IG host. The best solution to obtain a very fast update on these continuous controls is to use a shared memory connection between the GUI and the host. Unfortunately, restricting the GUI to only run in shared memory to the host is not the best solution, especially if we have the capability to use more than one machine or if the interface resides on another machine such as a personal computer (to accept input from the FastTrak device or from joysticks).

Therefore, if we transmit the whole GUI state whenever a change occurs, then the concept of the communication class should still remain. It is possible to transfer the state structure between processes over the network through shared memory with no difference on the structural level. It is then up to the host to maintain the current and previous GUI states to monitor for specific changes in the vehicle simulation. Furthermore, this needs to be done so that the host can send error and change information back to the GUI.

However, a better solution than sending the whole state is to just send the information which has changed. To do this, we can precede each message with an operation code, which will determine the type of message being sent. For example, an accelerator message can have operation code 1, a steering message can have operation code 2, and a fog message can have operation code 23. Certain state variables still need to be kept by the host for smoothing out continuous changes, but monitoring the state for changes in the GUI no longer is a function of the host.

The connection between the host simulation and the GUI is unicast. Therefore, naming the communication protocol as a GUI state PDU is a misnomer, but it seems to fit and it keeps the DIS

naming structure. It is impossible to establish this communication on the same network as the DIS port, so they must exist on different port bindings and thus not communicate through Entity Service. It will not interfere with normal DIS simulation operation.

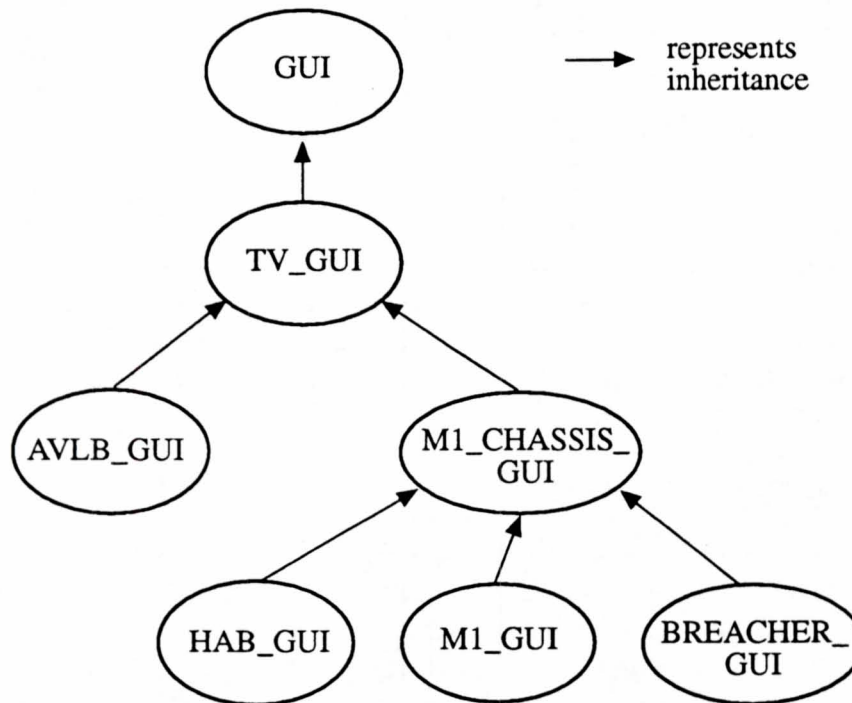
13.14.4 Communication Description

The GUI and the host will send message back and forth via the communication channel. This channel allows either ethernet packet transmission or shared memory communication. This communication is encapsulated and hidden from the calling process, which would be the GUI and the host. Other types of communication may be allowed in the future, but this will not affect the design at all. The communication channel is denoted in Figure 98 as the small dark rectangle. It is a part of the communication interface which merely instantiates the channel as a low level interface to the other process.

13.14.4.1 The GUI Hierarchy

An object-oriented approach will be taken with the design of the whole simulation, not just the GUI or simulation host. The design of the GUI must meet the criteria given in the preceding sections. A class object must communicate to the communication interface clearly and easily. An example class diagram is given below. This diagram represents four vehicles which may eventually be modeled in the Tracked Vehicle Simulator. The inheritance is based on preliminary information. We do not have any specific information about the Heavy Assault Bridge (HAB) or

the Grizzly Breacher.



0703-6939

Figure 99. Sample GUI Class Diagram

The current design can be coded up into C++ classes as follows:

```
class GUI
{
private:
    GUI_ci *ci;

public:
    virtual void show( void );
    virtual void hide( void );

    // this makes calls to the ci, which may use callback func-
    tions
    virtual void checkEvents( void );
};

class TV_GUI : public GUI
{
private:
    TV_GUIci *ci;
```



```

public:
    virtual void show( void );
    virtual void hide( void );

    // this makes calls to the ci, which may use callback func-
tions
    virtual void checkEvents( void );

    void processAccelerator(float accelerator);
    // other process functions defined here
};

class AVL_B_GUI : public TV_GUI
{
private:
    AVL_B_GUIci *ci;

public:
    virtual void show( void );
    virtual void hide( void );

    // this makes calls to the ci, which may use callback func-
tions
    // from the receiveMessage() function
    virtual void checkEvents( void );

    void processTransmission(AVL_B_trans transmission);
    // other process functions defined here
};

```

13.14.4.2 The Communication Interface Hierarchy for the GUI

A similar object-oriented approach to the design of the GUI can be taken for the communication interface. For the interface which sends and receives message from the GUI, the class hierarchy would resemble the one shown in the current design. The class descriptions follow:

```

class GUIci
{
public:
    // callback methods implemented by GUI client to process
incoming messages
    void receiveMessage(GUI* theGUI);
};

class TV_GUIci : public GUIci
{
public:
    void sendAccelerator( float accelerator );
    void sendBrake( float brake );
    void sendSteering( float steering );

    // overrides parent's receiveMessage function
    void receiveMessage(TV_GUI* theGUI);
};

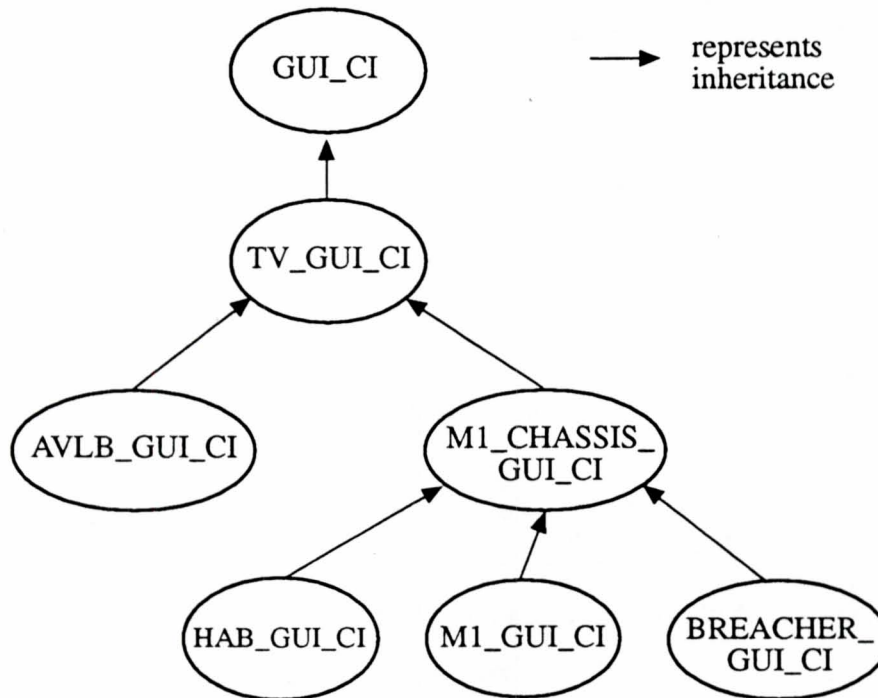
```

```

class AVLB_GUIci : public TV_GUIci
{
public:
    void sendTransmission( AVLB_trans transmission );
    // others are defined here

    // overrides parent's receiveMessage function
    void receiveMessage(AVLB_GUI* theGUI);
};

```



0703-6940

Figure 100. Sample Communication Interface Class Diagram

13.14.4.3 The Communication Interface Hierarchy for the Host

This hierarchy will need to resemble and communicate with the inheritance hierarchy of the simulation or the IG host. The simulation host's hierarchy will be very similar to the one shown above in both of the previous diagrams.

13.15 GUI Protocol

13.15.1 GUI Service

The only flaw in the preceding section is the difficulty in maintaining a previous and current GUI state, and then monitoring the changes. This could be a burden on the simulation host, which is something we should strive to avoid at all costs. Therefore, the following has been recommended to deal with this problem while retaining the concept of the GUI state and the abstracted communication protocol between the host simulation and the GUI.

The detailed part of this protocol resides with the simulation host. While we do not wish to place the constraint of shared memory communication between the GUI and host, it should be okay to do it for another process spawned by the host itself. We can start a new process in the host which communicates via shared memory to the host itself. This process, called the GUI Service, then sends and receives GUI state PDUs from the GUI process.

It is the GUI Service's responsibility to read the current state and monitor it for changes from the previous state which was received. It must then transfer only the changes to the vehicle simulation via shared memory constructs. The figure below shows a complete diagram of the communication between the three processes. The description of this diagram and further details of the GUI Service follows.

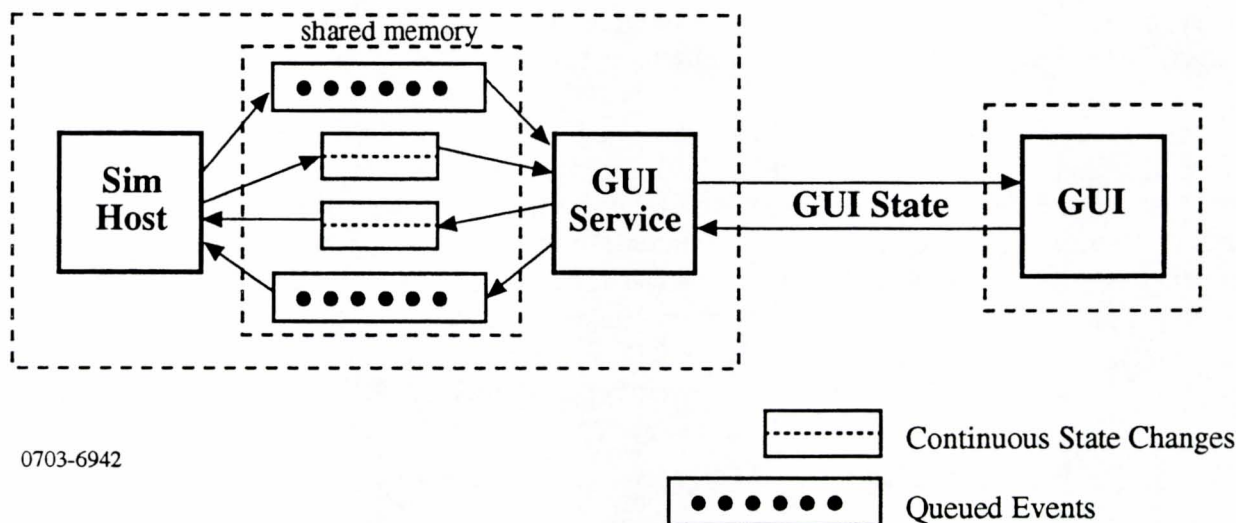


Figure 101. GUI State and the GUI Service

An important operation which the GUI Service process must also perform is the transformation of data from the GUI to the simulation host. The GUI may have values for the steering, for example, which range from -5.0 to +5.0. These values may be meaningless to the simulation host while keeping things much simpler in the GUI. It is therefore the GUI Service's responsibility to transform these values into something more useful to the simulation host. This applies for any of the controls which the host deems necessary.

The GUI Service and simulation host communicate by using shared memory structures. The boxes for continuous state changes and queued events depict these data structures. Two types of data transference have been identified: continuous and event.

13.15.2 Continuous State Data

This type of data refers to GUI control states which are continuously changing or at least possess the possibility to be modified in response to user input demands. Such controls are the steering and throttle controls, for instance. Others may depend on the vehicle model being employed by the simulation host.

Although it is unlikely that the GUI will exist on a separate machine and processor from the simulation host, the possibility does exist. Continuous data transfer will suffer greatly from the implementation, but that cannot be helped. Restricting the GUI to be on the same machine will avoid this problem, but that is, of course, not the appropriate solution.

To attempt to speed up transfer from the GUI to the simulation host for continuous data transfer, the GUI Service process will attempt a pseudo-dead reckoning algorithm to make the input more continuous. While it may be possible in the user interface to transfer from one extreme state (far left on the steering control) to another (far right) it should not be possible in the real world. It may also occur that the state changes are not received until this extreme condition is met. Therefore, the GUI Service plays a vital role in eliminating such a large step as would occur in this situation. It instead dead reckons the values which the simulation host will then read at its own rate, which is much faster than the GUI to GUI Service transfer rate.

The continuous data block is shown in Figure 101 by the rectangles with the thin dotted line. Notice that there is one block for each data path, from the GUI Service to the simulation host and back again. The GUI sends controls to the host, but sometimes it may not be possible to perform certain operations of the vehicle at different times. For instance, if the transmission is in park, then it would be impossible to move the vehicle forward. Thus, it should be possible for the simulation host to tell this to the GUI through the other continuous state data path that such a thing is impossible. Merely by resetting the values of the variables in shared memory can accomplish this objective. The GUI Service will notice the change and send the updated GUI PDU to the interface.

13.15.3 Event Driven Data

All other input which is not continuous can be called event driven. Examples of this include shifting gears, raising or lower an articulated part, detaching or attaching an articulated part, and firing a round. The user wishes a specific function of the vehicle to occur.

The principle difference between event driven input and continuous input is that event driven input is not so dependent on time latency. If the delay between shifting to neutral and actually being placed in neutral is three frames or one tenth of a second (for instance), then the user will probably not notice or even care. The goal of a driver-trainer simulator may require immediate processing of every command of the user, but the goal here is functionality. Furthermore, it is the

continuous which is the most important. Steering away from an oncoming missile while engaged in a dog fight with an enemy fighter is very important and response should occur immediately. One tenth of a second will not make much of a difference when opening the hatch.

The event driven shared memory queues are shown in Figure 101 by the rectangles with the thick dotted line. Similar to the continuous data paths, there is one queue to denote state changes from the GUI Service to the simulation host and vice versa. If a particular state of the vehicle cannot be reached for some reason, but the user attempts to reach that state, then it would be the simulation host's responsibility to inform the user. When an event is read which is impossible to process, a corresponding event is placed on the output queue to be read by the GUI Service. This event is then transformed back into GUI specific data and an updated GUI PDU is then sent. As you can see, the GUI Service operates very much like the entity and Terrain Service.

14.0 ESIG Host

14.1 Introduction to the ESIG Host

This document briefly describes the ESIG Host that was developed as part of the Dynamic Terrain (DT) Project at the Institute for Simulation and Training (IST). During 1993, engineers from Evans & Sutherland worked with IST to make changes to an ESIG 2000 image generator (IG) and the software in order to show the feasibility of networked dynamic terrain on a currently fielded IG. The successful demonstration of this approach at I/ITSEC '93 included an ESIG 2000 on a network with several workstations running various portions of the dynamic terrain system, with terrain changes appearing on both the workstations and the IG. The ESIG Host software was the component of the Dynamic Terrain system that controlled the IG. Note that for many reasons the ESIG Host software is not included in the prototype release. However, this document is provided to generally describe the ESIG Host software.

The basic concept behind the host is to respond to DIS events from the network and dead reckoning, and to send out appropriate opcode packets to the ESIG 2000. In particular, the following functions are supported:

- All DIS entities are displayed with articulated parts
- Detonation PDUs are displayed as explosion animations
- All terrain changes in DT Attribute PDUs are converted to DT opcodes
- A stealth is included in order to fly around the database on the ESIG

14.2 Analysis

This section describes how the ESIG Host reacts to each type of DIS PDU supported (Entity State PDU, Detonation PDU and DT Attribute PDU). The DT Attribute PDU is used to change attributes of terrain including elevation. For each elevation to be changed, the position and elevations are contained within the PDU. Note that the ESIG Host uses the Entity Service and the Terrain Service (see the corresponding documents) in order to receive PDUs in the form of messages (described more in the same documents). Furthermore, note that the ESIG Host communicates with the ESIG asynchronously and ignores all IG return packets (to simplify communication). Ideally, the ESIG Host should listen to return packets for errors.

14.2.1 Entity State Message

When each Entity State Message is received from the Entity Service, the entity ID contained within it is checked against all current entities. If it is not found, the ESIG Host assumes that a new entity has been created and marks it as new. Whether found or not, the Host will store the DIS type, position and orientation data of the entity for use in updating the IG. After all new states for entities have been read, the entities are checked to see if they were updated. Currently, if any entity has not been updated from the Entity Service after three rounds of updating, the entity is considered to have timed-out. In this case, an unbind opcode (3040) is sent to the IG.

During the update phase, the ESIG Host will send bind opcodes (3040) to the IG for each new entity to connect the model with a motion system. The positions and orientations are then sent to the IG by position opcodes (3041) for the primary model and secondary submodels.

14.2.2 Detonation Message

When a Detonation Message is received, the munition type is used to determine which ESIG model and animation sequence should be used as an explosion. The burst descriptor within the message contains the location where the detonation occurred. The ESIG Host uses this information to bind and position the model at the location specified, and to load and start the explosion animation sequence.

14.2.3 DT Attribute Message

When a DT Attribute Message is received, the Host uses the bounding box of the update within the message to find an axes-aligned bounding box. The new bounding box is used to requery the database and the new elevations are sent to the ESIG. To simplify the code, there are no checks to see which posts within the bounding box actually changed.

15.0 Simulation Host

15.1 Introduction to Simulation Host

15.1.1 Background

The Simulation Host was developed as a replacement for the simple simulators originally used to develop the Dynamic Terrain (DT) architecture. The simulation host is based on two fundamental changes in approach from the original dynamic terrain simulators. The ideas used by the simulation host came about during the critical analysis of those original simulators done for the dynamic virtual environments project. This analysis focused on the assumptions previously used in conventional simulator design.

Two main problem areas were uncovered. One, there was a high degree of connectivity between the actual simulation and man-in-the-loop feedback systems, primarily image generation systems. In pursuit of smoother performance for image generators, the actual simulation is often coupled tightly with the image generator and cannot function to its fullest capacity. In addition, maintenance and upgrading of the code is more difficult because it incorporates multiple logical functions. This can make simulation code unnecessarily dependent on other sections of code in the program.

The second problem is simply redundancy. Since prior simulators shared the same basic functionality, modifying elements of a shared environment, there is a potential for sharing segments of code. When this common functionality is reprogrammed in every simulator, a great deal of work is being needlessly repeated. Again maintenance and upgrading are a problem because several pieces of code have to be given the exact same modifications, a very repetitive process.

15.1.2 Problem Statement

The simulation host has two fundamental requirements. It needs to be focused on simulation only, so that logically separate functions do not adversely affect performance. It also must be general enough to support varying functions. In other words, it must remain flexible and cannot be optimized for a single special case.

15.1.3 Solution

The first goal of the simulation host, separation of the actual simulation from how its results are displayed, is accomplished using the shared environment built under the dynamic terrain architecture three. The interface to the shared environment is common to all applications and must therefore be a part of the host. Using Entity Service and Terrain Service, the simulation host can share state and event information with other applications that also use the services. These applications can be used to display and analyze the effects of the simulation.

The simulation host maintains the data from the services in a local database of environmental information, which the simulator can read from and write into. The host maintains consistency with the services on behalf of the simulator.

Another common simulation component is also incorporated into the host. This is clock management, such as scheduling, setting alarms, and time step estimation. If you consider time as just another part of the environment, the simulation host can be seen as an environment manager that maintains information external to the simulation, with which the simulation can interact. The simulator is therefore removed from considering how this information is maintained, thus allowing changes to how the shared environment works without significantly affecting how the simulator runs. A nice by-product of the host is that it standardizes the storage and querying of environmental information for everything it "hosts," creating a generic interface to the environment that is specialized towards simulation.

At this point, there are a number of ways that simulators can diverge. Depending on what is being simulated (an object, a phenomena, a piece of a shared system, etc.) and whether it's an active or passive simulation, i.e. whether or not user input is directly required.

Currently, the focus of the simulation host is supporting man-in-the-loop vehicle simulators, so that is the type of simulator that was developed. It is worth noting that other types of simulators can utilize the simulation host to interact with the environment.

While there is commonality to how active vehicle simulations operate, much of the data structures used are specific to particular vehicles. To handle this, object oriented hierarchies of objects were used in a generic vehicle simulation structure. The common facets of different logical data elements were abstracted into parent objects, with particular children setup to support the requirements of individual vehicles.

However, even with this design for data objects, there are still two areas of simulation that can vary, even for one particular vehicle. First is the user interface, which could be run as a graphical user interface with a program like FORMS, or an input device (such as a joystick, a spaceball or a mock-up of actual controls), a simple terminal interface that reads from a keyboard, or a network interface that is used to communicate with a remote machine. How user control data is generated, or feedback returned for that matter, is independent of how it is stored and transmitted within the simulator. This allows new user interfaces to be added to the simulator without affecting how the rest of the simulator operates.

The other variable part of a simulator is how the simulation is mathematically modeled. Different approaches can easily simulate the same vehicle, so the problem becomes one of giving different mathematical models a consistent way to plug into the simulation. Since the host maintains state and event information for the simulation, it's simply a matter of giving some arbitrary math model a way to change that environmental information and read in new changes to the environment.

The vehicle simulation needs to have a broad enough scope so that any particular user interface and math model can be supported.

For this work, the only type of user interface that is run is a FORMS panel, which is executed into a child process so that the graphical frame rate cannot affect the mathematical simulation rate.

The mathematical models used for the vehicles are based on the mobility model research that is explained in the mobility discussion in the Breacher reference.

15.1.4 Constraints/Assumptions

The only real constraints on the simulation host are that it needs access to Entity Service and Terrain Service. While it is possible to run the host with locally generated environmental data, without the services there is no way to view the results of the simulation. Remote processes such as the Dynamic Terrain Visualizer or the IG Host can then be used to help drive the vehicle and see the effects on other parts of the environment.

The current simulation is dependent on FORMS for providing a graphical user interface. This is currently a SGI-only program, although it is being ported to new platforms as Xforms. However, it is easy to convert the simulator to use another interface that is less machine specific.

15.2 Analysis

The basic problem designing the simulation host is to identify the redundant components in current dynamic terrain simulators, remove those components from the simulators and place them into the host. In effect, one piece of code, the simulation host, must perform these shared functions in a generic fashion with a flexible interface, so that stripped down simulators can use the host for performing common functions. This all needs to be done without considering how the effects of the simulator will be displayed.

Early on the duties of the host as an environmental manager became clear. However, other functions that were in the host had to be removed and placed down at the simulator level. Due to the wide range of possibilities in a simulation, the original host was divided in two: a common host that manages the environment and a generically designed simulator that is capable of becoming different vehicles. This separation also allows for different types of simulators to use the capabilities of the simulation host, possibly reducing overlap in future shared environment players.

Once the vehicle simulator was separated from the host, it soon became evident that it needed dividing up even more. How data is stored in the simulation, how user data gets into and out of the simulation, and how the mathematical model is implemented are all separate issues. By setting up the storage and transmission of controller and instrument settings in a generic way, the user interface and the mathematical model can be somewhat separated from each other and managed independently. Once these two major elements were put in the simulator, how they operated became almost unimportant; the generic structure of the vehicle simulator was complete. Further refinement of the simulator below this level falls out of the scope of the simulation host and the generic vehicle simulator. Those two parts need to be left open so that future modules can be plugged in.

15.3 Implementation

15.3.1 Problem Statement

There is a lot of repetition involved in creating a new simulator because work that has been done before for other systems must be re-done for the new simulator. While it is possible to optimize a special case simulator more, a great deal of “unnecessary” work is required to add each new type of simulator, because the design and implementation process has to be done almost from scratch every time. This greatly reduces the flexibility because each new simulation is basically a whole new piece of software.

The premise of the simulation host was a desire to develop a common structure for use in different kinds of simulators. The hope was that more generically designed components could be used in different types of simulators, thereby reducing the amount of new code that needed to be written to make a new simulator. Using a combination of generic, common code and special case software targeted at the simulation, overall workload and time required, is reduced while still allowing the simulator to perform all of the special functions needed for simulating “whatever” correctly.

15.3.2 User's Guide

There are only two areas of concern for someone running the simulation host, setting up the configuration files beforehand and using the graphical user interface while running. Running the simulation host requires a group of seven configuration files. These files specify how the simulation is configured in general, how it performs interoperation, which vehicle to simulate, how that vehicle should be configured, and how the user interface is setup. Once the program has started, the user controls the vehicle and receives feedback through a graphical user interface. This interface also controls program status and termination.

15.3.2.1 Configuration Files

As mentioned before, there are currently seven configuration files that are used to properly setup the simulator. These files can be divided into three groups; portal files, vehicle files, and the master configuration file.

The master configuration file is read by the program as a command line argument. All of the other files depend (either directly or indirectly) on the master file. In other words, multiple disjoint sets of files be kept and only the ones called for through the master file will be used. This makes it easy to setup and choose configurations on the fly. There is no default master configuration file so if one is not provided, the simulator cannot function. Similarly, there are no defaults for any of the other configuration files pointed to by the master file, so they also have to be directly specified and must also exist.

Currently the master configuration file is very simple. It contains the names of two other configuration files, the portal file and the vehicle model file. One example of this file is provided below.

It is meant for an M1A1.

```
# The Simulation_Manager's mlal configuration file.

# The name of the environment portal's configuration file.
environment_portal_file environment_portal.cfg
# The name of the general host's configuration file.
general_host_file mlal_host.cfg
```

The environment portal file mentioned above is used to specify how the program logs onto the services and manages the data it receives from them. Currently there is only one setup, although more may be added as different portal scenarios are added. The standard portal file currently in use is listed below.

```
# The Environment_Portal's standard configuration file.

# The name of entity portal's configuration file.
entity_portal_file entity_portal.cfg
# The name of terrain portal's configuration file.
terrain_portal_file terrain_portal.cfg
```

Currently these two files only contain the logon information required to connect to the Entity Service and the Terrain Service. Without this information, the program cannot successfully logon to either service. Both of these files are listed below.

```
# The standard configuration file for Entity_Networking.

# The identifying name used to confirm logons.
application_name Simulation_Manager
# An estimation of how frequently updates will be requested.
interface_cycle_rate 5
# The shared memory area used to initially connect to the service.
control_shared_memory_key 0x00004000
# The type of coordinates that will be used to communicate states.
coordinate_frame PERFORMER
# How the data time stamps will be measured.
absolute_time_stamps TRUE

# The standard configuration file for Terrain_Networking.

# The identifying name used to confirm logons.
application_name Simulation_Manager
# An estimation of how frequently updates will be requested.
interface_cycle_rate 5
# The shared memory area used to initially connect to the service.
control_shared_memory_key 0x00005000
```

The other file specified in the master configuration file was the general host configuration file. This file specifies what type of the simulation that the host will drive and how the simulation will be configured. Again an M1A1 file is used as an example.

```
# The General_Hosts's mlal configuration file.
```



```

# The name of the particular host.
type_of_host MlA1
# The name of the interface manager's configuration file.
interface_manager_file mlal_interface.cfg
# The name of the vehicle manager's configuration file.
vehicle_manager_file mlal_manager.cfg

```

These last two files configure the interface and model to a specific entity. The interface manager file is currently very simple, it only renames the interface. Currently only one interface exists for each entity, but since the potential exists for more, the interface is named.

```

# The Interface Manager's standard mlal configuration file.

# The name of the particular interface.
interface_name MlA1_FORM

```

The last file, the model manager configuration, sets all of the identification values that the entity needs to cooperate with other entities on the network. This file does not currently contain any information regarding how the model operates or starts up, although it may be added to this file in the future.

```

# The Vehicle Manager's standard mlal configuration file.

# The name exercise the entity will participate in.
exerciseID 1
# The entity's id number within the exercise.
entity 777
# The classification fields of the entity.
kind 1
domain 1
country 225
category 1
subcategory 1
specific 1
extra 0
# The entity's default appearance.
appearance 1
# The entity's dead reckoning algorithm.
drAlgorithm 3

```

15.3.2.2 User Interface

Currently there is only one user interface provided for each vehicle. This is a graphical user interface developed using FORMS. The interface is a FORMS panel containing controllers and instruments representing the controllers and instruments of the actual vehicle. A user simply manipulates the panel in logically the same way as they would use the real controllers.

Steering is represented by a lateral slider, the center position is dead ahead steering with left and right motion set to represent turning the handles. The center button just below the slider precisely re-centers the steering.

Two more sliders just below steering represent the throttle and the main break. At the bottom, they are fully released and at the top, they are fully depressed.

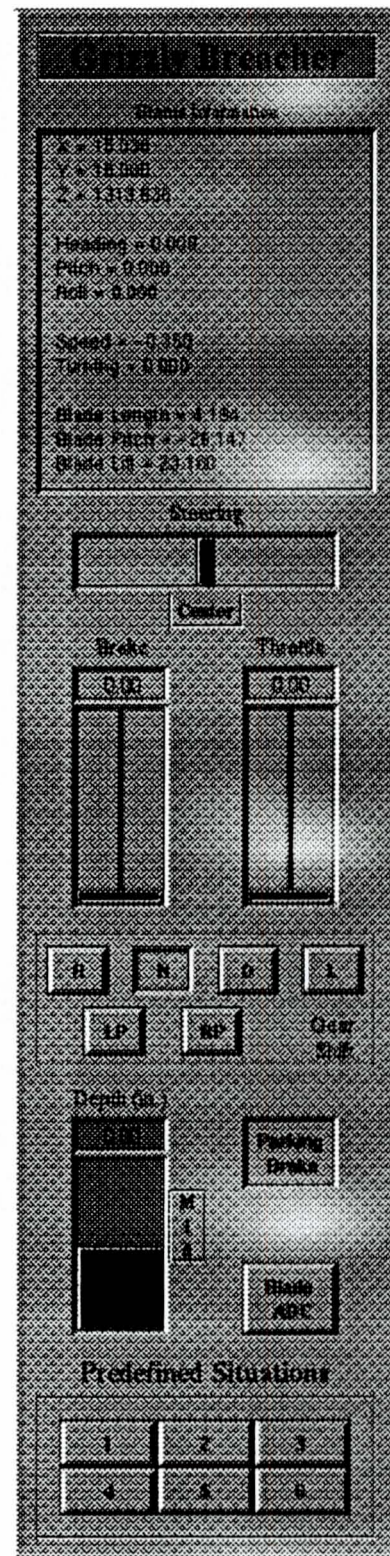
A set of buttons represents the transmission slots. One for each standard gear; reverse, neutral, drive, and low, as well as pivoting. The buttons must be pressed in order, passing by each adjacent one in the proper order.

One button represents the parking brake. It should be released before attempting to change gears and drive.

Two other special controllers on this panel are for the breacher in particular. The slider represents the requested blade depth of the digging mechanism. The button activates the automatic digging system that attempts to dig at the depth specified by the slider.

The status panel at the top represents the state information of the vehicle as it is being sent out by the vehicle model. This may be replaced with realistic instruments in the future.

The buttons at the bottom are leftovers kept for compatibility with another system.



15.3.3 Software Design

The general layout of all of the classes is shown in the following booch diagrams. The runtime layout of the objects is consistent, regardless of which type of vehicle is being simulated. All objects that are vehicle dependant are placed into the structure as abstract objects. The actual, vehicle dependant objects are then constructed at runtime depending on the precise configuration called for in the configuration files.

15.3.3.1 Top Level Simulation

The primary object in the system is the `Simulation_Manager` object. This object is instantiated within the `main()` file and drives the entire simulation. It contains five other objects within itself that constitute working parts of the simulation. It is responsible for properly configuring these five objects at the beginning of the simulation. Then it continues driving each object until it determines that the simulation should end or needs to be forcibly ended. Two of the objects it has are used solely for management purposes. The remaining three objects make up the core of the actual simulation.

One of the management objects used by the simulation manager is `Signal_Catcher`. This object is responsible for managing and redirecting signals from the operating system. These signals are divided into two categories: error signals, such as bus errors and segment faults, and shutdown signals, such as kill or quit. If a shutdown signal is received, the signal catcher simply makes a note of it and waits for the simulation manager to check back at the end of its next cycle. This way the simulation manager has a chance to cleanly close up all parts of the system, which can be especially helpful with `sproc'd` processes and interprocess communication established. If an error signal is detected, the signal catcher notifies the simulation manager immediately, begins shutdown, and prevents additional errors. The simulation manager will clean up whatever it can, but if another error is detected, the signal catcher will simply call for an immediate exit of the process.

The other management object used by the simulation manager is the `Alarm_Clock`. This object keeps track of the passage of time by using alarm signals from the operating system. The alarm clock sets alarms to whatever frequency is requested and maintains a counter of all the alarms that have gone off. At the end of each cycle, if no new alarms have come in, the alarm clock sleeps the process until a new one arrives, keeping the process from consuming unnecessary CPU time. If, however, there are alarms waiting to be serviced, then the simulation manager restarts immediately and attempts to process them. If undue alarms routinely build up, the alarm can be reset to a tighter interval.

The main simulation object is the `General_Host`. This object is responsible for the vehicle dependant parts of the simulation. The general host is actually an abstract object. It serves as a placeholder in the structure of the system. At runtime, one of the children of the general host (such as the `M1A1_Host`, the `Breacher_Host`, or the `Bulldozer_Host`) is actually constructed and run. The host object is just that, a host for all of the vehicle dependant code.

The next major simulation object is the `Shared_Environment`. This object holds all of the environmental data in the system. In effect, it acts much like an interpreting buffer or filter, that trans-

fers environmental data from one part of the simulator to another. One side it services the general host, taking its output and queries and preparing them for use by the environment portal. It also takes new information from the environment portal and prepares it for use by the host. The object does not really store very much information, but instead acts as a communicator, translating data cycle by cycle.

The last major object is the `Environment_Portal`. It handles all communication with the services and thereby, with the network. Basically, it is responsible for sending all newly generated environmental data in the shared environment out to the services as well as querying for new information needed by the shared environment in its next cycle.

All three of the main simulation objects have subordinate objects that they use to carry out their responsibilities. These objects are discussed in the next three sections.

15.3.3.2 Networking Objects

There currently two main objects used by the environment portal to communicate with the services, one per service. In the future, as more services are added, corresponding portal objects will be added here.

The connection to the Entity Service is maintained by `Entity_Networking`. This object hosts an entity service client interface with which it exchanges information. Currently, entity portal has two responsibilities. Its primary responsibility is sending out the current state of hosted vehicles. Entity portal checks the shared environment for local entities, reads their current state and identification, and transcribes that information properly into an entity state message that the client interface can understand. Its secondary responsibility is checking for detonations that have damaged the vehicle. It puts the type of damage, if any, into the shared environment for the host to detect and process.

The connection to the Terrain Service is similarly maintained by `Terrain_Networking`. This object hosts a terrain service client interface with which it also exchanges information. Terrain portal also reads and writes at both ends. It begins by checking the shared environment for patches of terrain that have been modified. If any exist, it puts the change into an attribute message and sends it to the client interface. Afterwards, it checks for new queries, i.e. patches of terrain that the host wants new, up to date information on. It queries the client interface for terrain data inside the queries and then forwards the information on into the shared environment.

15.3.3.3 Environmental Objects

The shared environment is also composed of two main objects, its environmental databases, one for storing entity information and one for storing terrain information. More may be added as new types of data are available.

Entity information is maintained by the `Ghost_Database`. This object maintains the current state of all entities hosted by the system, which is now currently limited to only one. This is done using an `Entity_Ghost` object, a simple passive object that holds state and damage information. The ghost state is updated by the host each cycle and then read by the environment portal. Vice versa

for the damage. The ghost is simply used to establish a standard for interchanging information between the host and the portal. An entity ghost is comprised of several objects including Spatial_Vector and Orientation objects as well as Identification and Damage objects.

Terrain information is likewise maintained by the Patch_Database. It keeps track of a series of Terrain_Patch objects. A terrain patch has two uses. Primarily, it stores actual terrain attribute information for use in changing or reading terrain. In addition, it can be used to setup queries for new terrain information. Again, the patch is a passive object that simply acts as an interchange standard between the host and portal. A terrain patch is composed of layers of objects that can be dynamically assembled to meet the needs of the host. Among these are the Attribute_Grid, the Row and Column objects, the Field_Extents.

15.3.3.4 Host Objects

The general host is the most complicated part of the system, because it is different for each type of entity simulated by the system. The discussion below deals with all host objects at an abstract level only. Since functionality is consistent for each object that inherits from the abstract objects, the system design can still be discussed.

There are actually four objects contained within a general host, two active objects and two passive ones. The active objects are the Vehicle_Manger and the Interface_Manger. The passive ones are the Controllers and the Instruments. The host sets up the controllers and instruments and then uses them to exchange information between the vehicle manager and the interface manager. It allows the two active objects to update themselves and each other each cycle.

The Controllers object stores the state of each controller accessible to the driver. As new inputs are generated in the user interface, the state of the controllers is updated and can then be read by the vehicle manager every cycle.

In a similar fashion, the Instruments object is updated by the vehicle manager each cycle. Its values are read by the user interface and turned into feedback for the driver.

The Interface_Manager is responsible for exchanging information between the driver and the rest of the system. This is currently done by *sproc'ing* a child process that can run at a rate independent of the main mathematical simulation loop. This is done because the frame rate of a graphics based interface can adversely affect the simulation time cycle used in mathematical models. The child process runs a graphical user interface using a GUI_Manager and an encapsulated object that holds FORMS interface calls. These two objects actually put a GUI up on the screen and process the commands that come from it. This includes making sure that controller inputs are possible and that instrument output is reasonable. The interface manager handles the time difference between the two processes and maintains the data in Controllers and Instruments.

The last major object in the system is the Vehicle_Manger. This object drives the actual mathematical models that simulate vehicle behavior. The vehicle manager is responsible for maintaining the data in Controllers and Instruments, as well and communication with the shared environment through the host. It prepares all information in a form that the math models can

understand, freeing the models from being concerned with the design of the simulation system. In effect, it acts as an object plug, where new math models can be plugged in and used right away, without having to develop a whole new simulator each time. The exact number and type of objects contained within a vehicle manager vary depending on how the math models were programmed. All current vehicles use a single object to represent the mathematics of the simulation.

15.3.4 Programmer's Guide

Most of the major design elements are explained in the prior section. The one remaining programming issue to discuss is the true advantage of the simulation host, its modularity. Since the host is designed in separable pieces, it's possible to replace one part without effecting the others, so long as the interfaces are kept the same. In addition, the host is designed to be expanded, such as perhaps adding a fluid portal and a fluid database to the system in the future. However, the primary advantage is that new vehicle based objects can be added by simply adding a new child to the inheritance from the abstract vehicle objects, thereby expanding the library of available vehicles.

Dynamic Terrain Utilities

This section discusses several utilities that aid in the configuration of a DT simulation. They are useful prior to a simulation exercise to test system configuration.

16.0 Dynamic Terrain Database Formatter

16.1 Introduction to the Dynamic Terrain Database Formatter

16.1.1 Problem Statement

A simple means for creating Dynamic Terrain Database (DTDB) files is provided by the DTDB Formatter, *perf2dtdb*, program. The tasks for this program are reading files in through the flexible IRIS Performer Application Programmer's Interface (API) libraries and writing out DTDB configuration, data, and attribute files. Another means of converting these files would be to create them by hand. However, this method is usually far too tedious and difficult. Also, it does not provide an easy means of visualizing the data as it is created.

16.1.2 Solution

Thus, *perf2dtdb* was developed. A FORMS Graphical User Interface (GUI) was designed as a front end for the program. FORMS is a sophisticated GUI designer and API toolkit for Silicon Graphics, Inc. (SGI) machines, written by Mark Overmars. It is public domain software. For more information, please refer to the FORMS library manual.

The code for *perf2dtdb* was written in C++, though no object-oriented classes were developed for the program. Some structures were created, and these were done with C++ in mind.

perf2dtdb does not provide visual feedback of the database itself. Other programs already exist to help view the DTDB in either wireframe or shaded views. One of these is the Visualizer. The image generator (IG) could also be used, though that may provide too much power for just visualizing the database.

The database files that the DTDB formatter reads can be visualized through the original modeling package. An example (and most common) database file would be one created with MultiGen Flight. Thus, MultiGen or ModelGen could be used to see how the original database looks.

16.2 Analysis

For such a simple project, an analysis was not necessary. If a more sophisticated conversion utility is desired, then this section might be required. However, some thought prior to coding showed that IRIS Performer can read many different file formats and a simple conversion from Performer to the DTDB format would help greatly. The file reading functionality provided through Performer are also updated by either SGI or by the original modeling package vendor (for instance,

MultiGen), so we do not have to continuously maintain the format reading code. The details of these formats and the implementation of the program are given below.

16.3 Implementation

16.3.1 Problem Statement

A Dynamic Terrain Database is specified through the use of two or more files. At least two are required, however. They are the configuration file and the elevation data file. The configuration file specifies the following parameters:

- **elevation surface representation:** the internal mathematical surface representation used to store the elevation data. The DTDB implements terrain via mathematical surfaces rather than merely grid points or polygons and polygon meshes. More information about the DTDB can be found in another document.
- **temporary surface representation:** while processing the data, a temporary or *transition* surface must be created. The type used is defined here.
- **texture scale parameters:** the DTDB conveniently provides texture coordinates for the user. However, the user may set the texture scaling parameters which define how many times a texture is repeated in both the *s* and *t* directions. Note that the texture is *not* defined by the DTDB, which is the desired functionality.
- **extents:** the extents of the database are given by four points: p1, p2, p3, and p4. They must be specified in clockwise order (it is important to note that this is contrary to normal coordinate specification). The reason for the clockwise ordering is that it allows the data to be organized right-to-left and top-to-bottom. Although they are not required to be axis aligned by the DTDB, they **should** be axis aligned for *perf2dtdb* to work probably. Bad things would happen otherwise.
- **number of attributes:** a DTDB may consist of more than one layer of data. Other layers, not including the elevation layer, may be created and used. An example attribute layer is thermal data.
- **elevation data file:** specifies the DTDB data file used for the elevation data.
- **attributes:** a set of four parameters may be specified for each extra attribute. These must be in order and grouped together. They are the attribute key, attribute name, attribute data file, and the attribute data surface representation.
- **attribute key:** gives the current attribute key, numbered from 1 to the maximum number of attributes.
- **attribute name:** this is just a convenient method for specifying a character description for the current attribute data.
- **attribute data file:** specifies the DTDB data file used for the current attribute data.
- **attribute data surface representation:** determines the surface representation for the current attribute data.

The data file has two more parameters to define:

- **extents:** this is merely a copy of the DTDB configuration extents. Unexpected results may occur if they do not match. Only Marty knows why this was even included.
- **number of posts:** the resolution of the data file is given by the number of posts in the

s and t direction. These posts range out over the extents so one more post should be given than the number of resolution units. For example, if the extents are 0,0 to 10,10 (axis aligned) and the resolution required is 1 in both directions then the number of posts should be 11,11.

Both a sample configuration file and its corresponding data and attribute files are given in a later section.

16.3.2 User's Guide

Using this program is quite simple. No configuration files are needed, though DTDB configuration files may be given on the command line. Just type in "perf2dtdb" and you will see the following window:

Figure 102. *perf2dtdb* Main Window

A DTDB configuration file may be given with the "-c" option. A space is required between the "-c" and the configuration file name. Only one may be given and only the last one specified takes effect. The FORMS interface will be updated to reflect the file. Anything specified after the last command line option is a data file to be read in through IRIS Performer's library routines. This means that MultiGen Flight files, Coryphaeus Designer's Workbench files, and other files may be

read in using the command line parameters.

16.3.2.1 Modifying the DTDB Configuration

The configuration file parameters discussed in a previous section may be modified through the FORMS interface. Easy methods are provided for updating the values for each parameter. A user already familiar with the FORMS interface objects will have no trouble using this program. In fact, the inexperienced user is recommended to read through parts of the FORMS library manual, because not all options will be given here.

The **elevation surface representation** and the **temporary surface representation** are shown at the top of the window (the top meaning just below the two logos). Both of these parameters can be changed using a similar FORMS object, the *choice* menu. If the left mouse button is pressed over the object, then the next option is chosen and displayed in the space provided. If the right mouse button is pressed then a pop-up menu appears and the user may freely choose from all the options listed. This last method is useful for showing all the options. For both parameters, Figure 102 shows that "UNRBS Bicubic" is chosen. The choices for surface representation are as follows:

- *UNRBS Bilinear*: bilinear uniform, nonrational B-Spline.
- *UNRBS Bicubic*: bicubic uniform, nonrational B-Spline.
- *Bitmap*: bitmap or grid.
- *Normal Triangle*: polygonal data structure of triangles, not meshed.

The current **configuration file name** is just below the surface representation choice objects. And the **elevation file name** is given just below that. These two parameters are also similar so they will be discussed together. An *input* object is provided for each file name. The user may press the left mouse button to be able to modify the input field directly. Alternatively, the user may press the **browse** button with the left mouse button to be able to browse through the directory structure of the disk. Unless the browse operation is canceled (by the user) the chosen file name will appear in the input field. If the file name is too long or the user deems the file path unnecessary, then the path may be stripped from the file name by using the **strip path** button. The most recent file name, directory, and pattern are remembered for each parameter. This includes the attribute file name and database file name (through the *load* button) described later.

The **texture spacing** parameters may be changed through a couple of *input* objects shown in the middle of the window on the left hand side. They are floating point values so the user may be more precise after the decimal point.

The **extents** of the database are given through four coordinates: P_1 , P_2 , P_3 , and P_4 . The user may modify each of these coordinate individually even though it is recommended to axially align the extents. In a future version, this may be restricted by only allowing the modification of P_1 and P_3 , corresponding to the lower left and upper right extents. In fact, this reference frame is the one that *perf2dtdb* expects when it queries the terrain for data file generation. A useful function is provided through the **Extents** button just to the right of the y coordinates. When this button is pressed, then the program calculates an axially aligned bounding box of the currently loaded terrain and sets the values in P_1 through P_4 accordingly. Note that the terrain data may not entirely

cover the extents, so personal modification of the coordinates may be necessary. If no terrain data has been loaded then the extents are zeroed.

After the configuration data is to the user's specification, then the **Write Cfg** button may be pressed to write the configuration file to disk. The file name is given in the previously defined *input* object. When the button is pressed, the label of the button will change to **Writing** and the button will stay depressed until the operation is complete. No interaction can take place until the writing of the file has finished, although most keyboard or mouse events will be remembered. If there is an existing file of the same name, it will be **overwritten**, so care is recommended.

16.3.2.2 Modifying the Data

The **elevation surface representation** choice object directly specifies how the elevation data is interpreted by the DTDB, but that information is stored in the configuration file, not in the data file. Only the database extents and the number of posts in the *s* and *t* directions are stored in the data file. The database extents for the data files (elevation and all attribute files) are the same as the configuration file extents.

The **number of posts** in the *s* and *t* directions are shown near the middle of the window on the left hand side, just above **texture spacing**. The two *input* objects are integer only, so the user may not use floating point numbers. In fact, it is impossible to enter a floating point value for the number of posts. It is **very important** to understand the coordinate system used by *perf2dtddb*. The number of posts is along the vector formed by $\overrightarrow{P_1P_4}$. fig shows the coordinate system more clearly. When a

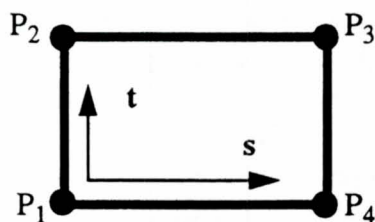


Figure 103. DTDB Coordinate System

configuration file is read from disk as specified on the command line, the elevation data file is also opened and the number of posts is read from the file. The *input* objects shown in Figure 102 are updated accordingly.

Database files loaded in through the IRIS Performer library might not be elevation specific. It may be difficult to visualize certain attributes using elevation, but it makes for the easiest conversion. The data loaded in may be used for some or all of the attributes. The functionality is provided to remove some of the data files and load in new ones for different attributes. The **Load** button at the bottom left corner of the main window gives the user the ability to load files during run-time. Remember that it is possible to load files via the command line. When this button is pressed, then a file directory is displayed and a file name may be selected. If a valid file is selected, then it will be loaded in, but the extents (the coordinate points P1 through P4) will not change until they are specifically altered.

A **clear** button is provided to immediately remove every data file that has been loaded. *perf2dtdb* will prompt the user to make sure that this operation should be carried out. Large data files could take a long time to load.

The **list** button opens another window, shown in Figure 102, which provides the user with a convenient *browser* object. The file names of any data files will be shown in the order they were

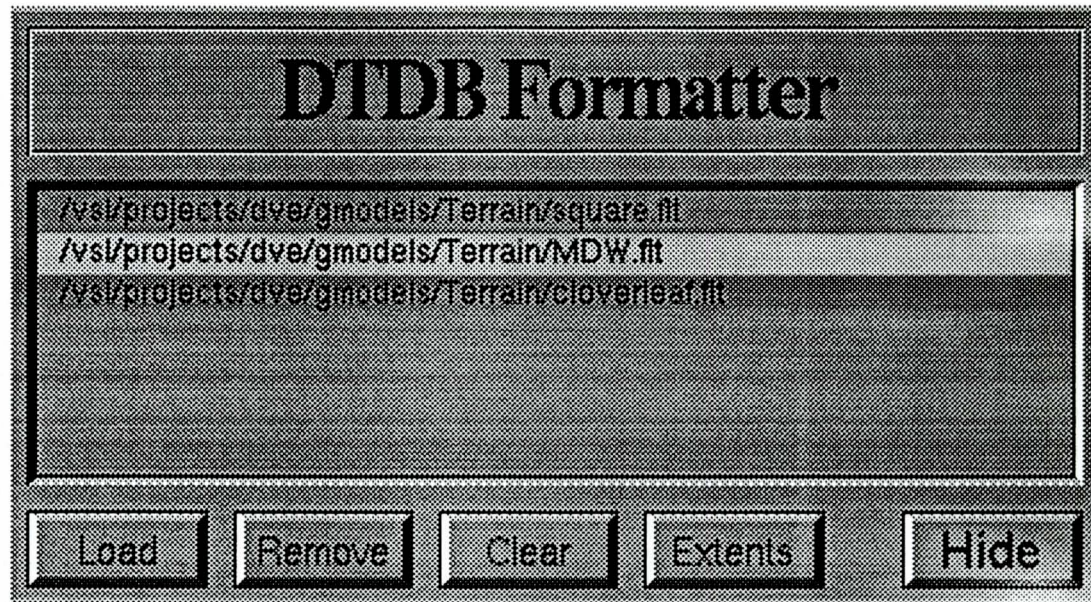


Figure 104. *perf2dtdb* Data File Browser

loaded in the middle of the window. If there are more files than can be shown then a scroll bar will appear on the left hand side, allowing the user the scroll back and forth to see and update all the data files. The left mouse button may be used to select a single file name at a time from the browser. This selection capability is used for the **Remove** and the **Extents** buttons. The **Load** button shown in the browser window functions the same as the previously defined **Load** button. Similarly, the **Clear** button from the browser window has the same functionality as the previously defined clear button. These two buttons were reproduced in the browser window for convenience. The **Remove** button removes the selected data file from the browser list and from memory. The **Extents** button in the browser window sets the coordinate points in the main window to the extents of the selected data file. Conversely, the **Extents** button in the main window uses all loaded data files in its computation. The **Hide** button closes the file browser window, but changes nothing in the system.

After the elevation data is to the user's specification, then the **Write Elev** button may be pressed to write the elevation file to disk. The file name is given in the previously defined *input* object. When the button is pressed, the label of the button will change to **Writing** and the button will stay depressed until the operation is complete. No interaction can take place until the writing of the file has finished, although most keyboard or mouse events will be remembered. If there is an existing file of the same name, it will be **overwritten**, so care is recommended. The procedure for querying the terrain data for elevation is simple. At each elevation grid point (post) the height is determined from the loaded data by shooting a ray downward from the sky and getting the Z value at

the intersection point between the ray and whatever it hits. If no intersection point is found, then an error message is printed and a value of 0.0 is used for that elevation post. Because these error messages may occur quite frequently, they are printed out to *stderr* (standard error, usually the console display). To understand why they might occur very frequently, it may help to look at a specific example. Let us say that the extents of the database are P1 (0.0, 0.0) to P4 (300.0, 300.0). With 1 unit resolution in both directions, there are 301 posts in both directions which makes 90,601 total posts and terrain elevation queries. If the user happens to make a mistake in either the extents or not loading all (or any) data files, then 90,601 error messages will appear. Obviously, this would crash the system if each message were printed to an individual window as some of the error message are done, so the errors are printed to *stderr* instead. It is highly recommended to initially use a small number of posts to make sure things are working correctly.

16.3.2.3 Modifying the Attribute Data

The attribute data information is shown in Figure 102 in the lower portion of the main window. It is sectioned off by a frame boundary. Most of the attribute information is stored in the DTDB configuration file, similar to the implementation with respect to the elevation file. The **number of attributes** used by the DTDB may be specified by the corresponding input object. The input will be clamped between 0 and the maximum number of attributes supported by the DTDB (currently, it is 20). All the other FORMS objects within the frame will be deactivated until the user sets the number of attributes greater than 0. As soon as a positive number of attributes are configured, they will be reactivated. Of course, the status of these objects will be determined appropriately upon the loading of the configuration file.

Only one of the attributes (if there are actually more than 1) can be shown on the main window. The **edit attribute input** object is provided for viewing and modifying other attributes. Upon starting *perf2dtdb* the first attribute will be shown or the objects will be set to reasonable defaults if there are no attributes.

The current **attribute file name** is shown just below the number of attributes input object. An *input* object is provided for the file name. The user may press the left mouse button to be able to modify the input field directly. Alternatively, the user may press the **browse** button with the left mouse button to be able to browse through the directory structure of the disk. Unless the browse operation is canceled (by the user) the chosen file name will appear in the input field. If the file name is too long or the user deems the file path unnecessary, then the path may be stripped from the file name by using the **strip path** button. The most recent file name, directory, and pattern are remembered for the attribute field, as discussed earlier.

The current **attribute name** may be modified through the input object shown just below the attribute file name object. It is recommended to use no spaces in the attribute name, because anything beyond the first word may not be remembered, though it will all be written to the file. The file format does not truly allow the use of spaces in the attribute name. A future version of the format may allow space, however.

The **attribute surface representation** is the last attribute configuration option. Please refer to the previous section on surface representations for more information.

Finally, the **number of posts** in the *s* and *t* directions may be specified individually for each attribute layer. This data is specific to the attribute data file and will only be used when the **Write Attrib** button is pressed. If a configuration file is specified on the command line, then an attempt will be made to open and read the header of every attribute file. Thus, the **number of posts** fields should correctly correspond to the files. The **Write Attrib** button operates the same way that the **Write Elev** button does, except that it uses the current attribute information, rather than the elevation information.

16.3.3 Examples

16.3.3.1 DTDB Configuration File

```
DTDB_CONFIG_FILE
temporary_surface_rep uniform_nonrational_bicubic_Bspline
texture_scale_s 25.0
texture_scale_t 25.0
p1_x -300.00
p1_y -300.00
p2_x -300.00
p2_y 300.00
p3_x 300.00
p3_y 300.00
p4_x 300.00
p4_y -300.00
number_of_attributes 1
elevation_data_file square.data
elevation_surface_rep uniform_nonrational_bicubic_Bspline
attribute_key 1
attribute_name Thermal
data_file thermal.data
surface_rep uniform_nonrational_bicubic_Bspline
```

16.3.3.2 DTDB Elevation Data File

```
DTDB_DATA_FILE
num_points_s 11
num_points_t 11
p1_x -300.00
p1_y -300.00
p2_x -300.00
p2_y 300.00
p3_x 300.00
p3_y 300.00
p4_x 300.00
p4_y -300.00
%
0.000      0.000      0.000      0.000      0.000      0.000
(the rest of the data portion has been deleted for viewing ease)
```


16.3.3.3 DTDB Attribute Data File

```
DTDB_DATA_FILE
num_points_s 15
num_points_t 15
p1_x -300.00
p1_y -300.00
p2_x -300.00
p2_y 300.00
p3_x 300.00
p3_y 300.00
p4_x 300.00
p4_y -300.00
%
      0.000      0.000      0.000      0.000      0.000      0.000
(the rest of the data portion has been deleted for viewing ease)
```

17.0 Surface Resampler

17.1 Introduction to Surface Resampler

This document describes the Surface Resampler and its usage. The Surface Resampler is one of the generation/interpolation utilities available in the Developer's Kit.

17.2 Analysis

The Surface Resampler simply requeries the database at the proper resolution. Therefore, the Dynamic Terrain Database (DTDB) does most of the work in that it can provide the data at the asked for resolution ("Introduction to the DynamicTerrain Database" on page 2). Note that it currently assumes a rectangle query is wanted.

17.3 Implementation

17.3.1 User's Guide

In order to execute the Surface Resampler, simply type "surface". The program will then ask you a series of questions:

- *DTDB configuration file.* This is the name of a DTDB configuration file that contains all the different attributes in the database.
- *Key of attribute to modify.* This is the unique key (from the DTDB configuration file) of the attribute to be modified.
- *Filename to write out new data.* This is the name of the file that the new data should be written to (it is a DTDB data file, not a DTDB configuration file).
- *Bounding box.* This contains four values that represent the south-west and north-east points to use as a bounding box (only these points are written to the new data file).
- *New number of values.* This is the number of points wanted in the new database file both in the two directions.

The program then creates a new DTDB data file.

18.0 Fractal Resampler

18.1 Introduction to Fractal Resampler

This document describes the Fractal Resampler and its usage. The Fractal Resampler is one of the generation/interpolation utilities available in the Developer's Kit.

18.2 Analysis

The Fractal Resampler uses some basic knowledge of fractals in order to perturb a terrain surface. The fractals used are based on Brownian motion and are described in Peitgen et al.

18.3 Implementation

18.3.1 User's Guide

In order to execute the Fractal Resampler, simply type "fractal". The program will then ask you a series of questions:

- *DTDB configuration file.* This is the name of a DTDB configuration file that contains all the different attributes in the database.
- *Key of attribute to modify.* This is the unique key (from the DTDB configuration file) of the attribute to be modified.
- *Filename to write out new data.* This is the name of the file that the new data should be written to (it is a DTDB data file, not a DTDB configuration file).
- *Bounding box.* This contains four values that represent the south-west and north-east points to use as a bounding box (only these points are written to the new data file).
- *Initial number of levels.* This is the initial number of levels.
- *Initial number of points.* This is the initial number of points and is generally equal to $2^{(\text{initialNumberOfLevels})} + 1$.
- *Desired number of levels.* This is the desired number of levels.
- *Desired number of points.* This is the desired number of points and is generally equal to $2^{(\text{desiredNumberOfLevels})} + 1$.
- *Initial value for σ .* This is the value for σ which is the initial standard deviation.
- *Value for H .* This is the value for H which is between 0 and 1 and indicates a level of "noise." A value closer to 0 will cause a noisier database.

The program then creates a new DTDB data file.

19.0 Terrain Service Configuration Utility

19.1 Introduction to the Terrain Service Configuration Utility

19.1.1 Problem Statement

A simple means for creating Terrain Service (TS) configuration files is provided by the TS configuration file editor, *tscfg*, program. The task for this program is merely to provide an easy means of generating TS configuration files. Another means of creating these files would be to edit them by hand using a simple editing tool like *vi* or *zip*. However, this method is tedious and difficult.

19.1.2 Solution

Thus, *tscfg* was developed. A FORMS Graphical User Interface (GUI) was designed as a front end for the program. FORMS is a sophisticated GUI designer and API toolkit for Silicon Graphics, Inc. (SGI) machines, written by Mark Overmars. It is public domain software. For more information, please refer to the FORMS library manual.

The code for *tscfg* was written in C++, though no object-oriented classes were developed for the program. Some structures were created with C++ in mind.

19.2 Implementation

19.2.1 Problem Statement

The TS configuration file has the following components:

- **service rate:** the TS operates at a specific rate in iterations per second. It is timed by the system clock to go no faster than the specified rate, but, if the machine is bogged down, then the service might operate more slowly. (parameter: *dbservHz*).
- **DTDB configuration file:** a Dynamic Terrain Database will be loaded by the Terrain Service upon starting. The database it loads depends on the value supplied by this parameter in the configuration file. (parameter: *dtdb_config_filename*).
- **control shared memory key:** shared memory is used to communicate between the TS and the client applications using the TS communication interface. To use shared memory, a shared memory key must be agreed upon by both processes. The TS uses the base key supplied by this parameter for use in control access communication. This key address is not used for terrain data transfer, but for such control aspects of the service like logon and logoff requests. (parameter: *control_shmkey*).
- **next shared memory key:** specifies the base key address for client applications to attach to the shared memory. The actual address that a client must attach to is passed through the control shared memory area. But, the starting address is defined in the ser-

- vice configuration file through this parameter. (parameter: *next_shmkey*).
- **delta shared memory key:** after each logon by a client application, the base shared memory key (*next_shmkey*) is incremented by this parameter. (parameter: *delta_shmkey*).
- **number of UDP links:** terrain update packets are too large to be broadcast over standard ethernet. However, they can be unicast to specific hosts (connected machines). The hosts to which the TS sends terrain updates are specified in the configuration file. This parameter gives the total number of User Datagram Protocol (UDP) links, which are the ethernet links between the TS and other machines which will be able handle the updates. Each link must obviously be described in the same configuration file by three more parameters: link number, address, and port. (parameter: *number_of_udp_links*).
- **UDP link number:** specifies the number of the link the configuration file is currently defining. The number must be between 0 and *number_of_udp_links*-1. (parameter: *link*).
- **address:** gives the Internet host address of the machine. For the configuration file, this address must be specified by number and in network byte order. Host name resolution is not performed by the TS. (parameter: *address*).
- **port:** for UDP communication across the ethernet, a port must be specified. (parameter: *port*).

A sample configuration file is given in a later section.

19.2.2 User's Guide

Using this program is quite simple. No configuration files are needed, though TS configuration files may be given on the command line. When the user types in "tscfg," the window shown in Figure 105 will appear.

A TS configuration file may be given with the "-c" option. A space is required between the "-c" and the configuration file name. Only one may be given and only the last one specified takes effect. The FORMS interface will be updated to reflect the file.

19.2.2.1 Modifying the TS Configuration

The configuration file parameters discussed in a previous section may be modified through the FORMS interface. Easy methods are provided for updating the values for each parameter. A user already familiar with the FORMS interface objects will have no trouble using this program. The inexperienced user is encouraged to read through parts of the FORMS library manual, since all options will not be presented here.

Figure 105. tscfg Window

The Terrain Service **configuration file name** is modified through the *input* object or the **browse button** identified by "TS Config File." Also, the **DTDB configuration file name** is specified by the input object and browse button identified by "DTDB Config File." These two parameters are similar and will be discussed together. An *input* object is provided for each file name. The user may press the left mouse button to modify the input field directly. Alternatively, the user may

Terrain Service Configuration

TS Config File

DTDB Config File

Rate (Hz) Number of UDP Links

UDP Links

Edit UDP Link Port

Link Address

Shared Memory

Control Key

Next Key

Delta Key

press the **browse** button with the left mouse button to be able to browse through the directory structure of the disk. Unless the browse operation is canceled (by the user), the chosen file name will appear in the input field. The most recent file name, directory, and pattern are remembered for each parameter.

The **service rate** parameter may be changed through an *input* object shown in the middle of the window on the left hand side, just below the DTDB configuration file name. It is a floating point value so the user may be more precise after the decimal point.

Another *input* object is provided to set the total **number of UDP links**. As usual, press the left mouse button while the pointer is in this field to modify the number. The tab or enter key must be pressed before the number change takes effect. If the number of links changes from 0 to any number greater than 0, then the *UDP Links* section becomes activated. Conversely, when the number of links is set to 0, the *UDP Links* section is deactivated and all previously defined links are lost. To add a new link, just increment the number. Decrementing the number removes the last links, but leaves the first ones unchanged.

The current link, as shown in Figure 105 in the *input* object **edit UDP link**, can be deleted by pressing the **delete** button on the right hand side. The label of the button is red to readily distinguish it from the others. When this button is pressed, the user is prompted to verify that the cur-

rent link should be deleted. At this prompt, the delete operation may be canceled.

The user may change the *input* parameter for **editing the current link**. The **port** and **address** input objects will be updated according to the change in link number. The TS configuration file must have the IP address for the machine name, but the user may specify the host name address instead on the **link address input** object. *tscfg* will attempt to resolve the name through the name server. Any errors in the name will be reported to the user. If the name is resolved correctly, then the **link address** will be changed to the corresponding IP address. Only the first IP address found will be used.

The *input* objects for the **control key**, the **next key**, and the **delta key** all operate on standard C numerical conventions. Any number preceded by a "0x" will be treated as hexadecimal. Any number preceded by only a "0" will be treated as octal. All others are treated as decimal. Also, *tscfg* will always set the input objects to reflect numbers in hexadecimal form.

After the configuration data is to the user's specification, then the **Write** button is pressed to write the configuration file to disk. The file name is given in the previously defined *input* object. When the button is pressed, the label of the button will change to **Writing** and the button will stay depressed until the operation is complete. No interaction can take place until the writing of the file has finished, although most keyboard or mouse events will be remembered. If there is an existing file of the same name, it will be **overwritten**, so care is recommended.

The user can load previously defined configuration files from disk by pressing the **Load** button. When the button is pressed, the label of the button will change to **Loading** and the button will stay depressed until the operation is complete. No interaction can take place until the loading of the file has finished, although most keyboard or mouse events will be remembered. The entire FORMS interface will be updated to represent the newly loaded configuration file. Any errors in the file will be reported to the shell window where *tscfg* was started. Note that this button is in addition to the command line parameter "-c".

19.2.3 Examples

19.2.3.1 tserv_grail_crusader.cfg

```
TSERV_CONFIG_FILE
```

```
# Frame rate of Terrain Service
#
# Format:
#
#   tservHz n
#
#       where
#
#           n = how many times per second to process data
#
tservHz 40.0
```

```

# DTDB configuration file
#
# Format:
#
#   dtddbConfigFilename f
#
#       where
#
#           f = Filename of DTDB configuration file
#
dtddbConfigFilename dtddb.cfg

# Max clients
maxClients 5

# Shared memory keys
controlShmkey 0x5000
nextShmkey 0x9000
deltaShmkey 0x10

# Network Configuration
#
# Format:
#
#   netConfig t
#
#       where
#
#           t = Configuration type of service which is one of:
#               server (merge immediately, forward to all links)
#               client (don't merge immediately, forward to others)
#               relay <server_link_key> (don't merge immediately,
forward 'up')
#               distributed (merge client updates immediately, for-
ward to others)#
netConfig distributed

# Communication links
#
# Format:
#
#   link k i p
#
#       where
#
#           k = unique key for this link
#           i = IP address of machine
#           p = Port number to use for this link
#
link 0 132.170.194.38 4000

```

19.2.3.2 tserv_grail.cfg

```
TSERV_CONFIG_FILE

# Frame rate of Terrain Service
#
# Format:
#
#   tservHz n
#
#   where
#
#       n = how many times per second to process data
#
tservHz 40.0

# DTDB configuration file
#
# Format:
#
#   dtddbConfigFilename f
#
#   where
#
#       f = Filename of DTDB configuration file
#
dtddbConfigFilename dtddb.cfg

# Max clients
maxClients 5

# Shared memory keys
controlShmkey 0x5000
nextShmkey 0x9000
deltaShmkey 0x10

# Network Configuration
#
# Format:
#
#   netConfig t
#
#   where
#
#       t = Configuration type of service which is one of:
#           server (merge immediately, forward to all links)
#           client (don't merge immediately, forward to others)
#           relay <server_link_key> (don't merge immediately,
forward 'up')
#           distributed (merge client updates immediately, for-
ward to others)#
netConfig distributed

# Communication links
```



```
#
# Format:
#
# link k i p
#
#     where
#
#     k = unique key for this link
#     i = IP address of machine
#     p = Port number to use for this link
#
link 0 132.170.194.38 4000
link 1 132.170.194.36 4000
link 2 132.170.194.35 4000
link 3 132.170.194.34 4000
```

20.0 Fluid Service Configuration Utility

20.1 Introduction to the Fluid Service Configuration Utility

20.1.1 Problem Statement

A simple means for creating Fluid Service (FS) configuration files is provided by the FS configuration file editor, *fscfg*, program. The task for this program is merely to provide an easy means of generating FS configuration files. Another means of creating these files would be to edit them by hand using a simple editing tool like *vi* or *zip*. However, this method is usually far too tedious and difficult.

20.1.2 Solution

fscfg was developed to easily generate FS configuration files. A FORMS Graphical User Interface (GUI) was designed as a front end for the program. FORMS is a sophisticated GUI designer and API toolkit for Silicon Graphics, Inc. (SGI) machines, written by Mark Overmars. It is public domain software. For more information, please refer to the FORMS library manual.

The code for *fscfg* was written in C++, though no object-oriented classes were developed for the program. Some structures were created with C++ in mind.

20.2 Implementation

20.2.1 Problem Statement

The FS configuration file has the following components:

- **service rate:** the FS operates at a specific rate in iterations per second. It is timed by the system clock to go no faster than the specified rate, but if the machine is bogged down, then the service might operate more slowly. (parameter: *dbservHz*).
- **DTDB configuration file:** a Dynamic Terrain Database will be loaded by the Fluid Service upon starting. The database loaded depends on the value supplied by this parameter in the configuration file. (parameter: *dtdb_config_filename*).
- **control shared memory key:** shared memory is used to communicate between the FS and the client applications using the FS communication interface. To use shared memory, a shared memory key must be agreed upon by both processes. The FS uses the base key supplied by this parameter for use in control access communication. This key address is not used for fluid data transfer, but for such control aspects of the service like logon and logoff requests. (parameter: *control_shmkey*).
- **next shared memory key:** specifies the base key address for client applications to attach to the shared memory. The actual address that a client must attach to is passed

through the control shared memory area. But, the starting address is defined in the service configuration file through this parameter. (parameter: *next_shmkey*).

- **delta shared memory key:** after each logon by a client application, the base shared memory key (*next_shmkey*) is incremented by this parameter. (parameter: *delta_shmkey*).
- **number of UDP links:** fluid update packets are too large to be broadcast over standard ethernet. However, they can be unicast to specific hosts (connected machines). The hosts to which the FS sends fluid updates are specified in the configuration file. This parameter gives the total number of User Datagram Protocol (UDP) links, which are the ethernet links between the FS and other machines which will be able handle the updates. Each link must obviously be described in the same configuration file by three more parameters: link number, address, and port. (parameter: *number_of_udp_links*).
- **UDP link number:** specifies the number of the link the configuration file is currently defining. The number must be between 0 and *number_of_udp_links*-1. (parameter: *link*).
- **address:** gives the Internet host address of the machine. For the configuration file, this address must be specified by number and in network byte order. Host name resolution is not performed by the FS. (parameter: *address*).
- **port:** for UDP communication across the ethernet, a port must be specified. (parameter: *port*).

A sample configuration file is given in a later section.

20.2.2 User's Guide

Using this program is quite simple. No configuration files are needed, though FS configuration files may be given on the command line. The user will see the window shown in the figure below by typing "fscfg" at the prompt.

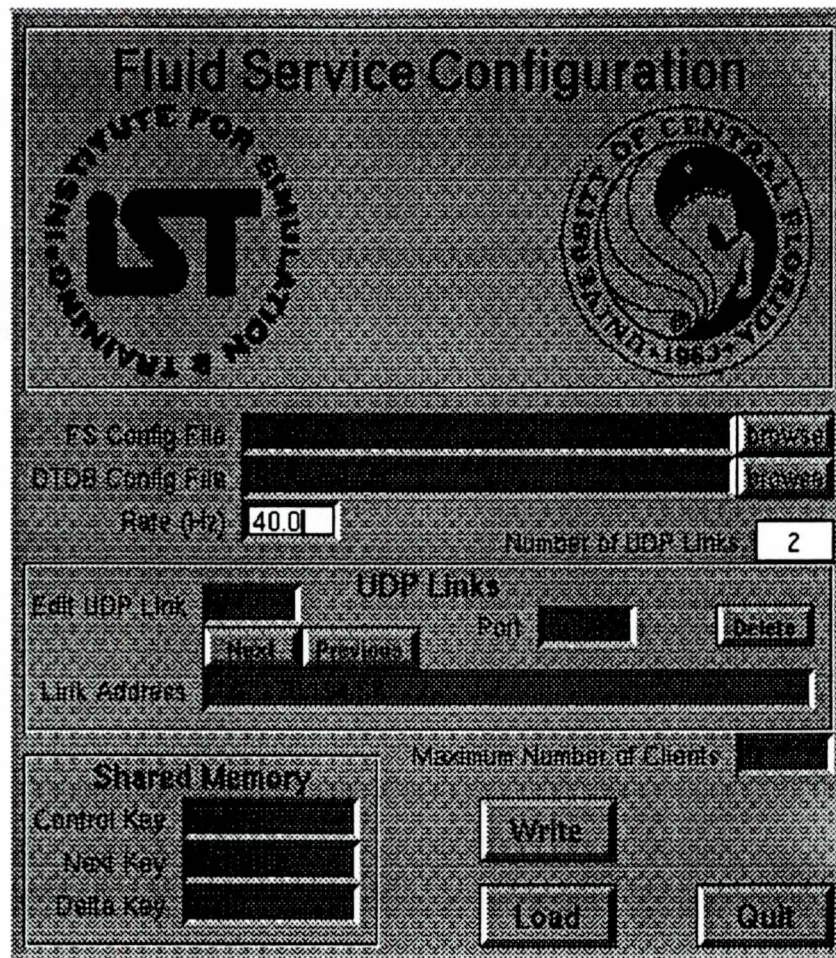
A FS configuration file may be given with the "-c" option. A space is required between the "-c" and the configuration file name. Only one may be given and only the last one specified takes effect. The FORMS interface will be updated to reflect the file.

20.2.2.1 Modifying the FS Configuration

The configuration file parameters discussed in a previous section may be modified through the FORMS interface. Easy methods are provided for updating the values for each parameter. A user already familiar with the FORMS interface objects will have no trouble using this program though it is recommended that the inexperienced user read through parts of the FORMS library manual, because not all options will be presented in this section.

Figure 106. fscfg Window

The Fluid Service **configuration file name** is modified through the *input* object or the **browse button** identified by "FS Config File." Also, the **DTDB configuration file name** is specified by the input object and browse button identified by "DTDB Config File." These two parameters are similar and will be discussed together. An *input* object is provided for each file name. By pressing



the left mouse button, the user will be able to modify the input field directly. Alternatively, the user may press **browse** with the left mouse button to browse through the directory structure of the disk. Unless the browse operation is canceled (by the user), the chosen file name will appear in the input field. The most recent file name, directory, and pattern are remembered for each parameter.

The **service rate** parameter may be changed through an *input* object shown in the middle of the window on the left hand side, just below the DTDB configuration file name. It is a floating point value so the user may be more precise after the decimal point.

Another *input* object is provided to set the total **number of UDP links**. As usual, press the left mouse button while the pointer is in this field to modify the number. The tab or enter key must be pressed before the number change takes effect. If the number of links changes from 0 to any number greater than 0, then the *UDP Links* section becomes activated. Conversely, when the number of links is set to 0, the *UDP Links* section is deactivated and all previously defined links are lost. To add a new link, just increment the number. Decrementing the number removes the last links, but leaves the first ones unchanged.

The current link, as shown in Figure 106 in the *input* object **edit UDP link**, can be deleted by pressing the **delete** button on the right hand side. The label of the button is red to readily distin-

guish it from the others. When this button is pressed, the user is prompted to verify that the current link should be deleted. At this prompt, the delete operation may be canceled.

The user may change the *input* parameter for **editing the current link**. The **port** and **address** input objects will be updated according to the change in link number. The FS configuration file must have the IP address for the machine name, but the user may specify the host name address instead on the **link address input** object. *fscfg* will attempt to resolve the name through the name server. Any errors in the name will be reported to the user. If the name is resolved correctly, then the **link address** will be changed to the corresponding IP address. Only the first IP address found will be used.

The *input* objects for the **control key**, the **next key**, and the **delta key** all operate on standard C numerical conventions. Any number preceded by a "0x" will be treated as hexadecimal. Any number preceded by only a "0" will be treated as octal. All others are treated as decimal. Also, *fscfg* will always set the input objects to reflect numbers in hexadecimal form.

After the configuration data is to the user's specification, then the **Write** button may be pressed to write the configuration file to disk. The file name is given in the previously defined *input* object. When the button is pressed, the label of the button will change to **Writing** and the button will stay depressed until the operation is complete. No interaction can take place until the writing of the file has finished, although most keyboard or mouse events will be remembered. If there is an existing file of the same name, it will be **overwritten**, so care is recommended.

The user can load previously defined configuration files from disk by pressing the **Load** button. When the button is pressed, the label of the button will change to **Loading** and the button will stay depressed until the operation is complete. No interaction can take place until the loading of the file has finished, although most keyboard or mouse events will be remembered. The entire FORMS interface will be updated to represent the newly loaded configuration file. Any errors in the file will be reported to the shell window where *fscfg* was started. Note that this button is in addition to the command line parameter "-c".

20.2.3 Example

20.2.3.1 fserv_crusader.cfg

```
FSErv_CONFIG_FILE

# Frame rate of Fluid Service
#
# Format:
#
#   fservHz n
#
#   where
#
#       n = how many times per second to process data
#
```

```

fservHz 40.0

# DFDB configuration file
#
# Format:
#
#   dfdbConfigFilename f
#
#       where
#
#           f = Filename of DFDB configuration file
#
dfdbConfigFilename dfdb.cfg

# Max clients
maxClients 5

# Shared memory keys
controlShmkey 0x6000
nextShmkey 0x10000
deltaShmkey 0x10

# Network Configuration
#
# Format:
#
#   netConfig t
#
#       where
#
#           t = Configuration type of service which is one of:
#               server (merge immediately, forward to all links)
#               client (don't merge immediately, forward to others)
#               relay <server_link_key> (don't merge immediately,
forward 'up')
#               distributed (merge client updates immediately, for-
ward to others)#
netConfig distributed

# Communication links
#
# Format:
#
#   link k i p
#
#       where
#
#           k = unique key for this link
#           i = IP address of machine
#           p = Port number to use for this link
#
link 0 132.170.194.37 5000
link 1 132.170.194.36 5000
link 2 132.170.194.35 5000

```


link 3 132.170.194.34 5000

21.0 Sources

21.1 Introduction to Sources

21.1.1 Problem Statement

A simple means of generating network packets is provided by the *sources* program. The task for this program is outputting Distributed Interactive Simulation (DIS) Protocol Description Units (PDUs) for entity states, detonations, and fires. Another means of sending these PDUs onto the network is to run a simulation host such as the Grizzly Breacher or the Tracked Vehicle Simulator (TVS). However, running the TVS may provide far too much power for the user's needs. Sometimes it is only desirable to obtain a single circling entity for testing or demonstration purposes. Also, the TVS does not output all network traffic, only AVLB entity states. For detonations and fires, the user would have to look elsewhere.

21.1.2 Solution

A FORMS Graphical User Interface (GUI) was designed as a front end for the program. FORMS is a sophisticated GUI designer and Application Programmer's Interface (API) toolkit for Silicon Graphics, Inc. (SGI) machines. It is public domain software. For more information, please refer to the FORMS library manual.

The code for *sources* was written in C++, but no object-oriented classes were developed for the main portion of the program. The Entity Service client interface and the Terrain Service client interface are both object classes, so it became necessary for *sources* to be coded for C++. Also, it was quite convenient and desirable to implement the entity state, detonation, and fire generation modules in an object-oriented language.

21.2 Analysis

A *module* is a term used to describe an encapsulated object program, which may be reused by other programs. The generators used by *sources* are all called modules, so they may be reused by other programs, as the name implies. The programmer should find this development useful in either modifying this code or adapting the modules for other applications.

A parent class, the DISModule, was created to encapsulate the logon and logoff to the Entity Service. The simple Booch class diagram is shown in Figure 108. The rate at which the application module is supposed to communicate with the Entity Service is passed to the DISModule through the constructor. The rate specifies the number of times per second that the module will attempt communication. The base shared memory key is also passed through the constructor method. The documentation on the Entity Service describes what is meant by the shared memory key. Communication refers to either outputting a network PDU or querying the service for some information. However, since none of the three modules implemented for the *sources* program needs to query

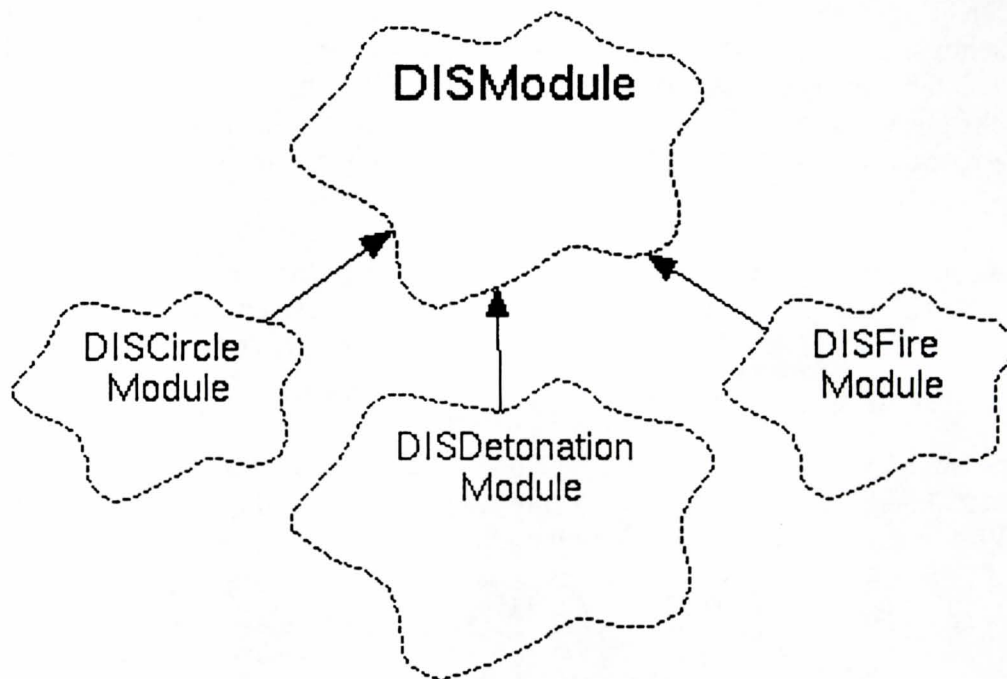


Figure 107. Module Booch class diagram

the Entity Service, only output communication will be used.

Another feature which might be implemented for this application is to allow a group of modules to execute the same main program. These modules could easily be managed by some sort of dynamic list structure which would allow an unspecified number of circle, detonation, or fire modules. However, this feature has not been implemented. To get more than one vehicle entity to circle on the network, the user must execute the *sources* program for each entity.

21.3 Implementation

21.3.1 Problem Statement

Sources can generate a single entity state PDU for any DIS vehicle type and a detonation and fire PDU for any DIS munition type. The entity state PDU will possess correct values for dead reckoning the vehicle and will handle any of the dead reckoning parameters. However, algorithm four is recommended for dead reckoning. This algorithm specifies that position is dead reckoned with global linear velocity and constant global linear acceleration and orientation is dead reckoned with constant angular velocity. The specifics of this algorithm and others can be found in the appropriate DIS standards manuals or in the documentation on the Entity Service and the Entity

Service Client Interface.

Detonations can be generated one at a time at the user's discretion, or at a certain rate by the program. The first mode is called SINGLE mode and a detonation is outputted when the user presses a button. The two other modes are RANDOM and CONTINUOUS. Both of these modes output detonation PDUs at the given rate. RANDOM places each subsequent detonation at a random location within the given circular area. CONTINUOUS merely outputs the same detonation at the same location.

Fire PDUs are generated only when the user presses the button on the fire interface window. The fire module does not logoff the Entity Service until the window is closed via the *Cancel* button.

21.3.2 User's Guide

Using this program is quite simple. No configuration files are needed. Also, no command line parameters are required. Just type in "sources" and you will see the following window, shown in Figure 108. When the program first starts, it will not logon to the Entity Service. Only four but-

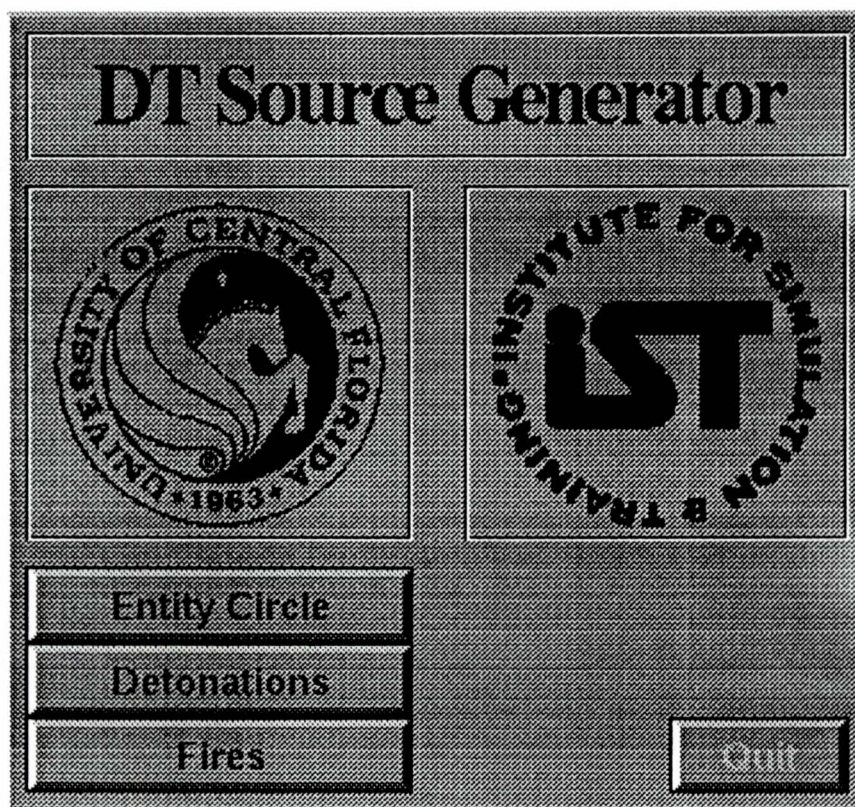


Figure 108. DT Source Generator Main Window

tons are available on this window: *Entity Circle*, *Detonations*, *Fires*, and *Quit*. The functionality of *Quit* should be obvious. The Quit button is for the whole program. If engaged, all open windows will be closed and all modules will be shut down and deleted. The *Quit* button is for the whole program.

The other three buttons will open up the corresponding window appropriate to the named module. Note that the modules are not actually created, and thus not logged on to the Entity Service, until told to do so. Also, if the window has been hidden, then these buttons will merely reopen the window for the user's modification.

21.3.2.1 Entity Circle

When the user presses the *Entity Circle* button, the window shown in Figure 109 will be opened.

Figure 109. Entity Circle Window

As previously mentioned, the circle module is not yet allocated until the *OK* button is pressed. The user should examine the information in the window before continuing with the logon event. Any changes must occur before the module is created and logs on to the service. Likewise, after the module is logged on to the Entity Service, changes in the interface window will only take effect after the module has been deallocated (logged off) and reallocated (logged back on again). All three modules connect to the Entity Service and thus that part of the interface, shown in the upper left corner of Figure 109, which will be detailed in section 21.3.2.4 titled "Entity Service" later in this document.

The dead reckoning algorithm is shown in the *DR algorithm* input object in the upper right corner of the window. The value for this object is not checked so the user must be careful not to specify a bogus parameter. A future specification for DIS may allow a wider range for dead reckoning.

The entity type is given in the middle of the window by the seven parameters which define the DIS type: *Kind*, *Domain*, *Country*, *Category*, *Subcategory*, *Specific*, and *Extra*. Another object is given just above the *Specific* and *Extra* input objects. This object is the *Type* choice object and allows the user to specify the entity type without having to remember all seven parameters. The DIS enumeration scheme for entities is actually quite large, so this is a reasonable feature. Currently, only a few of the entity types are available, but any programmer can easily add more if the need arises. In Figure 109, the current entity type chosen is an F18 fighter jet, which has the DIS specification shown in the window.

The *Center* of the circle, *Radius*, and *Speed* must also be given before the *OK* button is pressed. The units for these options are in meters for the center and radius and degrees per second for the speed. The entity will circle around the given origin center at the given radius and speed. Updates will be sent to the Entity Service at the previously specified rate.

If a *Land* based entity is to be generated, then sources will attempt to force it to follow the terrain. *Land* entities have a value of 1 for the domain (in the current DIS standardization, v2.0.3). Terrain clamping will only occur for entities whose primary mode of movement is by land; there is no clamping toggle button.

To clamp the specified entity to the terrain, *sources* must logon to the Terrain Service. At some point, the shared memory key to the service may change, so an input object is provide which will allow modification of the shared memory key used to attach to the Terrain Service process. This is shown in the figure as "TS Key." The default value, as shown in the figure, is 0x5000 in hexadecimal notation.

When using *sources* to generate entities, the user may find it necessary to change each entity's identification number. This ID is actually specified with three short integers: *site*, *host*, and *entity*. Generally, the *site* should define the location of the simulated entity (such as which network it originates from). The *host* specifies the machine. The last integer, *entity*, gives the actual entity ID, in case more than one entity originates from the node. *sources* gets the host name from the operating system, and sets the values for the entity ID by default. *site* is given the third IP address specification (which refers to the particular subnet or local area network) and *host* is given the machine address. For instance, if *sources* is run on a machine with the IP address 132.170.194.37, then the *site* will be set to 194 and the *host* will be set to 37. A number for *entity* was chosen at random.

The last three buttons are described in more detail in section 21.3.2.5 titled "OK, Cancel, or Hide" later in this document. They are similar for every module used by the *sources* program.

21.3.2.2 Detonations

When the user presses the *Detonations* button, the window shown in Figure 110 is opened. As previously mentioned, the detonation module is not yet allocated until the *OK* button is pressed. The user should examine the information in the window before continuing with the logon event. Any change must occur before the module is created and logs on to the service. Likewise, after the module is logged on to the Entity Service, changes in the interface window will only take

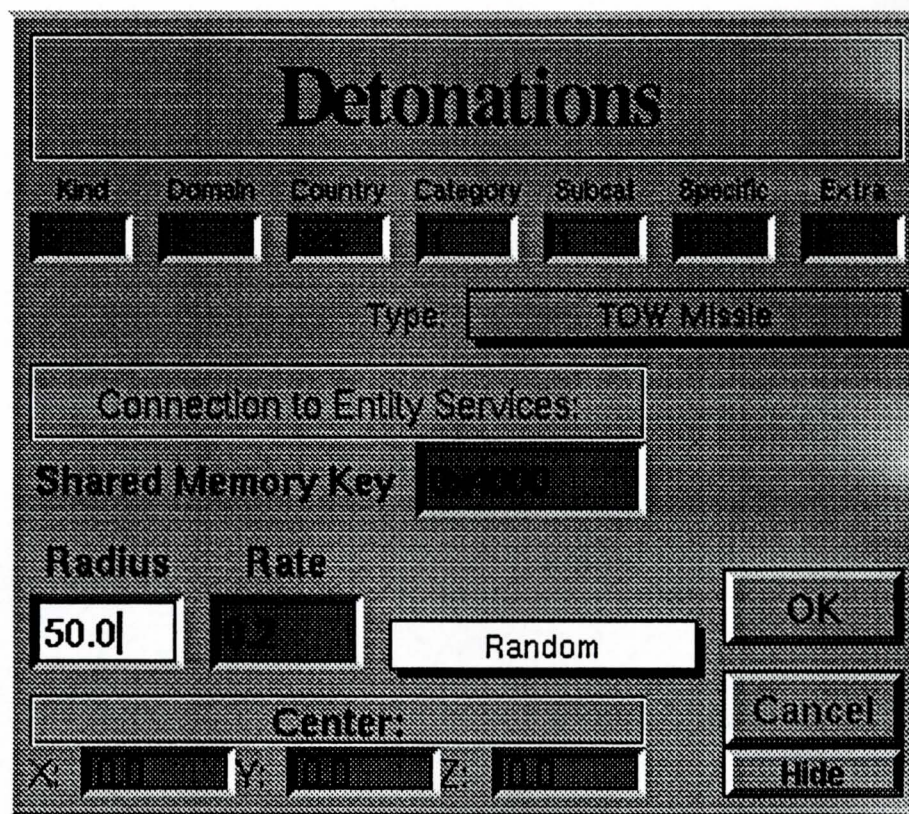


Figure 110. Detonations Window

effect after the module has been deallocated (logged off) and reallocated (logged back on again). All three modules connect to the Entity Service and thus that part of the interface, shown in the middle of Figure 110, which will be detailed in section 21.3.2.4 titled "Entity Service" later in this document.

Three types of detonation modes are supported: SINGLE, RANDOM, and CONTINUOUS. For SINGLE, a detonation PDU is sent only when the user presses the *OK* button. For the other two modes, a PDU is outputted according to the given *Rate*. The following paragraph describes a few more input objects which are necessary for these modes.

The *Center* of the circle and *Radius* must also be given before the *OK* button is pressed. The units for these options are in meters. The detonations will be generated around the given origin center and no further than the given radius in RANDOM mode. They will be generated at the given center for SINGLE and CONTINUOUS mode. The *Radius* is irrelevant for these last two modes. Updates will be sent to the Entity Service at the previously specified rate.

The detonation munition type is given at the top of the window by the seven parameters which define the DIS type: *Kind*, *Domain*, *Country*, *Category*, *Subcategory*, *Specific*, and *Extra*. Another object is given just below the *Specific* and *Extra* input objects. This object is the *Type* choice object and allows the user to specify the munition type without having to remember all seven parameters. The DIS enumeration scheme for munitions is actually quite large, so this is a reason-

able feature. Currently, only a few of the munition types are available, but any programmer can easily add more if the need arises. In Figure 110, the current entity type chosen is a TOW missile, which has the DIS specification shown in the window.

The last three buttons are described in more detail in section 21.3.2.5 titled "OK, Cancel, or Hide" later in this document. They are similar for every module used by the *sources* program.

21.3.2.3 Fires

When the user presses the *Fires* button, the window shown in Figure 111 opens. As previously

The screenshot shows a graphical user interface window titled "Fires". At the top, there are seven dropdown menus labeled "Kind", "Domain", "Country", "Category", "Subcat", "Specific", and "Extra". Below these is a "Type:" label followed by a text box containing "TOW Missile". Underneath is a "Connection to Entity Services:" label. This is followed by a "Shared Memory Key" label and a text box containing "000000". Below this are two labels, "Range" and "Quantity", each followed by a text box; the "Range" box contains "50.0". To the right of these text boxes are three buttons: "OK", "Cancel", and "Hide". At the bottom of the window is a "Location:" label followed by three text boxes labeled "X", "Y", and "Z".

Figure 111. Fires Window

mentioned, the detonation module is not yet allocated until the *OK* button is pressed. The user should examine the information in the window before continuing with the logon event. Any changes must occur before the module is created and logs on to the service. Likewise, after the module is logged on to the Entity Service, changes in the interface window will only take effect after the module has been deallocated (logged off) and reallocated (logged back on again). All three modules connect to the Entity Service; that part of the interface, shown in the middle of Figure 111, will be detailed in section 3.2.4 titled "Entity Service" later in this document. The Fire Module does not have a *Rate*, however. The rate is determined by how fast the user presses the *OK* button. The rate given to the Entity Service is 1.

The fire munition type is given at the top of the window by the seven parameters which define the DIS type: *Kind*, *Domain*, *Country*, *Category*, *Subcategory*, *Specific*, and *Extra*. Another object is

given just below the *Specific* and *Extra* input objects. This object is the *Type* choice object and allows the user to specify the munition type without having to remember all seven parameters. The DIS enumeration scheme for munitions is actually quite large, so this is a reasonable feature. Currently, only a few of the munition types are available, but any programmer can easily add more if the need arises. In Figure 111, the current entity type chosen is a TOW missile, which has the DIS specification shown in the window.

The *Range* of the Fire is specified by the corresponding input object. The functionality of this parameter is further defined by the DIS standard. The number of fires is given by the *Quantity* input object. This is useful for specifying more than one fire simultaneously, or at least very close together. For instance, a machine gun with a high rate of fire may have a *Quantity* value of 4 or 10.

The *Location* of the fire PDU must also be given before the *OK* button is pressed. The units for the *Location* option is in meters. The fires will be generated at the given origin location. PDUs will be sent to the Entity Service only when the user presses the *OK* button.

The last three buttons are described in more detail in section 21.3.2.5 titled "OK, Cancel, or Hide" later in this document. They are similar for every module used by the *sources* program.

21.3.2.4 Entity Service

Sources communicates with the service via IRIX shared memory structures and semaphores. This type of communication requires a key. The key specifies a location in memory where the shared memory structures and semaphores are located. Although more than one structure and semaphore is used for each program, only the base key is needed. In general, the default key should **not** be changed. If, for some reason, the Entity Service key has changed, this option has been provided. To change the key, place the mouse cursor in the input area and press the left button. The background color of the input bar will change to white. You may use standard C conventions for the key; i.e., any number starting with a 0x is hexadecimal, any number starting with a 0 is octal and all others are decimal. You **must** hit enter or tab after changing to key for the change to take effect. The Entity Service does not allow run-time changes.

Another option is the rate. *Sources* will send PDUs to the network at a specific rate. More importantly, this rate must be provided to the service at logon time. The maximum rate is determined by the Entity Service itself and is not available to the application program or to the user. To find out what this value is, look in the Entity Service window (when the Entity Service is started, the rate is printed out). The rate input area is similar to the key input, but C conventions are not allowed, only integer values may be used.

21.3.2.5 OK, Cancel, or Hide

The three modules have the same three buttons on the bottom right hand corner of the interface window. These buttons are *OK*, *Cancel*, and *Hide*. The *OK* button allocates the module if not yet allocated and logs on to the Entity Service. This button works differently for the Fire Module

because the Fire Module does not output Fire PDUs at a continuous rate. The same thing holds for the Detonation Module during SINGLE mode. For these circumstances, the module will be instantiated only on the first PDU outputted and will not be destroyed until the *Cancel* button is pressed. Otherwise, the *OK* button will stay depressed until it is pressed again. At this point, the module is deallocated and logged off the Entity Service, but the interface window is not closed.

To deallocate the module, log off the Service, and close the window, by pressing the *Cancel* button. This option may be executed at any time.

The *Hide* button merely closes the window. The module, if started, will still continue to operate and possibly output PDUs to the Entity Service and subsequently to the network. This is useful for generating entity states or detonations while reducing window clutter on the workstation's screen. Hiding the Fire Module window does not deallocate the module, but no fire messages will be generated.

21.3.3 Programmer's Guide

The Booch class diagram is shown in Figure 108 earlier in this document. As the programmer can see, only four object classes are implemented especially for the *sources* program. Of course, others are also used but they are detailed elsewhere, (such as the Entity Service Client Interface).

The main source file, *sources.c++*, has a simple algorithm. It creates the FORMS interface objects, displays the main window shown in Figure 108 and then loops on input from the GUI. In each loop, before the actual interface is checked, each module is updated (if it exists). The entity circle module is updated first, then the detonation module, and finally the fire module.

21.3.3.1 DISModule

The abstract parent class module is defined by the following class structure:

```
class DISModule
{
protected:
    EntityServiceClientInterface *escl;
    int rate;

public:
    DISModule( int _rate = 20, key_t key = 0x4000 );
    ~DISModule( void );

    virtual void sim( void ) = 0;
};
```

Note that although it is an abstract class (denoted by the pure virtual function *sim*), it does have a protected variable which is instantiated by the constructor and deleted by the destructor. The Entity Service Client Interface object is detailed in the appropriate documentation. The *rate* is stored in a protected variable for possible use by the subclasses.

21.3.3.2 DISCircleModule

The Entity Circle subclass is defined by the following class structure:

```
class DISCircleModule : public DISModule
{
private:
    static u_short entityID;

protected:
    EntityMsg msg;
    int first;
    double sec, accum;
    float radius, speed_d, speed_r, center[3];
    float startang, angle, xyz[3], hpr[3], acceleration, acc[3];
    float angvel[3], velocity, vel[3];

public:
    DISCircleModule( EntityType &type, int _rate = 20, key_t key =
0x4000 );

    virtual void sim( void );

    void setRadius( float _radius );
    float getRadius( void );
    void setSpeed( float _speed );
    float getSpeed( void );
    void setCenter( float _center[] );
    void getCenter( float _center[] );
    void setDRAlgorithm( int _radius );
    int getDRAlgorithm( void );
};
```

This class is quite a bit more complex than its parent. The private class variable *entityID* is used to maintain an increasing integer which will be incremented for every new entity. As an entity is generated, this variable will be incremented and if the circle module is deallocated and reallocated in the same *sources* session, any new circles generated will be incremented.

21.3.3.3 DISDetonationModule

The Detonation subclass is defined by the following class structure:

```
class DISDetonationModule : public DISModule
{
private:
    static u_short entityID;

protected:
    DetonationMsg msg;
    int first, go, freq;
    double sec, acc;
    float radius, det_rate, center[3];

    void explode( void );

public:
```



```

DISDetonationModule( int _rate = 20, key_t key = 0x4000 );

virtual void sim( void );

void setRadius( float _radius );
float getRadius( void );
void setRate( float _rate );
float getRate( void );
void setCenter( float _center[] );
void getCenter( float _center[] );
void setType( EntityType &_type );
void getType( EntityType &_type );
void setFrequency( int _type );
int getFrequency( void );
void start( void );
void stop( void );
};

```

Also, the enumeration for the detonation frequency is given by:

```

#define DETONATION_SINGLE      1
#define DETONATION_RANDOM     2
#define DETONATION_CONTINUOUS 3

```

These correspond to the SINGLE, RANDOM, and CONTINUOUS types discussed earlier in this document. It is important to maintain the relationship between this enumeration and the FORMS choice enumeration for the detonation frequencies.

The private class variable *entityID* is used to maintain an increasing integer which will be incremented for every new detonation. As a detonation is generated, this variable will be incremented and if the detonation module is deallocated and reallocated in the same *sources* session, any new detonations generated will be incremented.

For detonations to occur, the DISDetonationModule must be started with the **start** method. Likewise, it should be stopped with the **stop** method. Using these methods may be useful for other implementations of the detonation module. The **sim** method must still be called every simulation (program) loop.

21.3.3.4 DISFireModule

The Fire subclass is defined by the following class structure:

```

class DISFireModule : public DISModule
{
private:
    static u_short entityID;

protected:
    FireMsg msg;
    int go;
    float center[3];

public:
    DISFireModule( int _rate = 20, key_t key = 0x4000 );

```

```

virtual void sim( void );

void setRange( float _range );
float getRange( void );
void setQuantity( float _quantity );
float getQuantity( void );
void setCenter( float _center[] );
void getCenter( float _center[] );
void setType( EntityType &_type );
void getType( EntityType &_type );
void start( void );
void stop( void );
};

```

Obviously, this class is much less complex than either the Entity Circle or Detonation classes. The private class variable *entityID* is used to maintain an increasing integer which will be incremented for every new fire. As a fire is generated, this variable will be incremented and if the fire module is deallocated and reallocated in the same *sources* session, any new fires generated will be incremented.

For fires to occur, the DISFireModule must be started with the **start** method. Likewise, it should be stopped with the **stop** method. Using these methods may be useful for other implementations of the fire module. The **sim** method must still be called every simulation (program) loop.

22.0 Sinks

22.1 Introduction to Sinks

22.1.1 Problem Statement

A simple means for viewing network packets is provided by the sinks program. The tasks for this program are grabbing and printing out Distributed Interactive Simulation (DIS) Protocol Description Units (PDUs). Another means of viewing these PDUs would be to run the Image Generator (IG). However, running the IG may provide far too much power for the user. Also, the IG does not output all network traffic to the screen, and certainly does not print packet information.

22.1.2 Solution

Thus, *sinks* was developed. A FORMS Graphical User Interface (GUI) was designed as a front end for the program. FORMS is a sophisticated GUI designer and Application Programmer's Interface (API) toolkit for Silicon Graphics, Inc. (SGI) machines, written by Mark Overmars. It is public domain software. For more information, please refer to the FORMS library manual.

The code for *sinks* was written in C++, though no object-oriented classes were developed for the program. The entity service client interface and the terrain service client interface are both object classes, so it became necessary for *sinks* to be coded for C++.

22.2 Analysis

For such a simple project, an analysis was not necessary. If a more sophisticated PDU grabber is desired, then this section might be required. However, some consideration prior to coding showed that five PDU structures can be grabbed from the network. The details of these PDUs and the implementation of the program are given below.

22.3 Implementation

22.3.1 Problem Statement

Sinks will grab and print entity state, fire, detonation, attach, and hey you PDUs. Entity state PDUs are packets describing entities in the simulation. Entities relate to all moving or movable vehicles and other objects. Certain munitions will also generate entity states. An example of this kind of munition would be a TOW missile during its flight. The launch of a TOW missile generates a fire PDU. While the missile is in flight, an entity state is generated (per DIS standards). When the missile completes its mission and explodes either upon a given target or upon a miss, a detonation PDU is generated that describes the target, attacker, and munition types. This program

will also handle the attach PDU, which is an IST Visual Systems Lab prototypical PDU to show AVLB (Armored Vehicle Launched Bridge) launches and recoveries. When the AVLB deploys the bridge, an attach PDU is generated with a flag specifying that the bridge has been ejected. When an AVLB recovers the bridge, another attach PDU is generated which relates this fact. More information on attach PDUs and their necessity can be found in another document. Another prototypical PDU is the dynamic terrain PDU. When a terrain change is made by a simulation host, the change is sent out on the network. This prompts a "hey you" signal to the receiving hosts. Modifying terrain can take up a great deal of processing time, so the ability to read these terrain changes is given up to the host. Thus, the host may check for recent terrain changes by looking at the "hey you" signals. *Sinks* can print out all terrain changes as they are received by the Terrain Service.

22.3.2 User's Guide

Using this program is quite simple. No configuration files are needed. Also, no command line parameters are required. Just type in "sinks" and you will see the following window.



Figure 112. Sinks GUI operating window

When the program is first started, it will not logon to either the Entity Service or the Terrain Service. This way, you may run them in either order. If you are not logged on to one of the services, then the corresponding buttons and inputs for that service will be deactivated, i.e. the grabbing

functions for those PDUs will not be available until sinks attaches to the service. Entity states, fires, detonations, and attaches are all received from the Entity Service, and hey yous are received from the Terrain Service.

22.3.2.1 Entity Service

There are four options in the Entity Service area. The choice already mentioned is the logon button. This button is shown in Figure 112 on the bottom left hand area of the window. The name of the button changes, depending on whether or not you are logged on to the Entity Service. At first, the label of the button will be "Logon," denoting that you may logon to the service. After logging on, the button label will change to "Logoff."

Sinks communicates with the service via IRIX shared memory structures and semaphores. This type of communication requires a key. The key specifies a location in memory where the shared memory structures and semaphores are located. Although more than one structure and semaphore is used for each program, only the base key is needed. In general, the default key should **not** be changed. If, for some reason, the Entity Service key has changed, this option has been provided. To change the key, place the mouse cursor in the input area and press the left button. The background color of the input bar will change to white. You may use standard C conventions for the key; i.e. any number starting with a 0x is hexadecimal, any number starting with a 0 is octal and all others are decimal. You **must** hit enter or tab after changing to key for the change to take effect. If you are already logged on to the Entity Service, then you will be logged off and logged back on again. The Entity Service does not allow run-time changes.

Another option is the rate. Sinks will read PDUs from the services at a specific rate. More importantly, this rate must be provided to the service at logon time. The maximum rate is determined by the entity service itself and is not available to the attaching program or to the user. To find out what this value is, look in the Entity Service window (when the Entity Service is started, the rate is printed out). It is important to note that sinks will print out information at the minimum rate of both services. That is, the output rate r is computed by $r = \min(\text{entity_service_rate}, \text{terrain_service_rate})$. The rate input area is similar to the key input, but C conventions are not allowed, only standard integers may be used.

The last option is specifying the coordinate conversion type. Applications attached to the Entity Service may be happily oblivious to the coordinate frame(s) used by the DIS standard. The entity service client interface allows the client (sinks) to specify its own coordinate system. Upon login, four choice are available for this option: IRIS Performer, flat earth, geocentric, and ESIG. IRIS Performer is an SGI API developed for high performance rendering. This toolkit was used to create the IG and is the integral coordinate system for many of the applications we have developed in the DT project. Flat earth and geocentric coordinate systems are rarely used in this environment, but the capability for outputting those conversions are provided. The Evans and Sutherland image generators use yet another coordinate system. To change the conversion type, press the left button. Press down and hold the right button to get a pop-up menu listing all the choices.

22.3.2.2 Terrain Service

The buttons and inputs for the services are similar to the ones described in "Entity Service" on page 268. The base shared memory key for the terrain service is different, however. Also, it is generally accepted that reading terrain PDUs is much slower than reading other types of network traffic, so the Terrain Service rate is usually much lower. Currently, no conversion is provided in the terrain service client interface and is reflected in the available options shown in Figure 112.

22.3.2.3 PDU Buttons

For each PDU type, there are five buttons or inputs. In Figure 1, the first button on the left signifies whether that particular type of PDU will be outputted when it occurs. When a button is pressed, it will stay depressed until released. As shown in Figure 112, entity states will be outputted, but no other PDUs will be displayed. None of the buttons or inputs for each PDU type have any effect on the other PDU types.

The next two buttons to the right determine *where* the information is outputted. C and C++ programmers will understand the meaning of **stdout**. **Stdout** is short for *standard output* and identifies the console display. The standard output will be the shell window where sinks was started. You may instead give a file name. Although the output going to stdout to a disk file maybe redirected, it would be a difficult task to segregate the various PDU outputs. Thus, a simple means of specifying a file name is provided.

The next button, labeled *browse*, opens a window with a GUI directory and file browser, making it easy to set the file name of the output file for the given PDU type. You may also directly type in the name of the file in the input area at the far right. Unlike the input areas for the shared memory keys and rates for the services, you may type any character from the keyboard. Some keyboard commands are possible and these are described in the FORMS documentation. Even though a file name may be specified, the file will **not** be opened until the *file* button is depressed. It is important to note that files are opened in **append** mode. In other words, no file will be overwritten or deleted by this operation. All output will be appended to the file, so comfortably changing from *stdout* to *file* without destroying previous information is easily accomplished.

22.3.3 Example Output

The following two PDU packets were taken at a rate of 20 Hz from the Entity Service. The entity type is a Grizzly Breacher. The rest of the information in the PDU is described in the DIS standards documentation.

```
Entity=====
ID:          1 2 777
type:        1 1 225 3 9 0 0
alt type:    1 1 225 3 9 0 0
position:    0.000, 0.000, 0.000
orientation: -90.000, 0.000, 0.000
dead reckoning parameters:
  algorithm:  3
  linear velocity: 0.000, 0.000, 0.000
```



```

        linear acceleration: 0.000, 0.000, 0.000
        angular velocity:   0.000, 0.000, 0.000
force ID:      1
appearance:    1
capabilities:  0
articulated parts:
    number:    3
    part 0:
        change:      0
        attached to: 0
        type:        4109
        value:       0.000
    part 1:
        change:      0
        attached to: 0
        type:        4099
        value:       0.103
    part 2:
        change:      0
        attached to: 0
        type:        4141
        value:       -0.039

```

```

Entity=====
ID:          1 2 777
type:        1 1 225 3 9 0 0
alt type:    1 1 225 3 9 0 0
position:    0.000, 0.000, 0.000
orientation: -90.000, 0.000, 0.000
dead reckoning parameters:
    algorithm:      3
    linear velocity: 0.000, 0.000, 0.000
    linear acceleration: 0.000, 0.000, 0.000
    angular velocity: 0.000, 0.000, 0.000
force ID:      1
appearance:    1
capabilities:  0
articulated parts:
    number:    3
    part 0:
        change:      0
        attached to: 0
        type:        4109
        value:       0.000
    part 1:
        change:      0
        attached to: 0
        type:        4099
        value:       0.103
    part 2:
        change:      0
        attached to: 0
        type:        4141
        value:       -0.039

```

The next PDU is an example of a detonation. The detonation is from a TOW missile, as evidenced by the *burst type*. All aspects of the PDU are fully described in the DIS documentation.

Detonation=====

attacker ID: 0 0 0
target ID: 0 0 0
munition ID: 1 2 1
event ID: 0 0 0
velocity: 0.000, 0.000, 0.000
position: 10.414, 8.270, 0.000
burst type: 1 2 225 1 15 0 0
warhead: 1400
fuze: 1200
quantity: 1
rate: 0
entity pos: 0.000, 0.000, 0.000
result: 3
articulated parts:
 number: 0

Appendices

This section is a collection of several components that did not fit neatly into the previous sections. The first is a description of a minefield breacher simulation using mobility and vehicle dynamics. The second is a digging tool model with the flexibility to accomodate many digging surfaces. These digging surfaces are defined by the user with a geometric description. Both coordinate conversions and dead reckoning are methods used within the Entity Service (See "Introduction to Entity Service" on page 33.). The last section is a User's Manual for the IG Host (See "Introduction to the IG Host" on page 151.).

23.0 Mobility and Vehicle Dynamics

23.1 Introduction to Mobility and Vehicle Dynamics

23.1.1 Motivation for Mobility Studies

Mobility of tracked vehicles is an important issue from the design, simulation and training perspective. Terrain characteristics are important for off-road vehicle traction and mobility. Previously, it was difficult to include mobility calculations in man-in-the-loop simulations. Dynamic Terrain (DT) provided the capability to modify, represent and visualize the changing terrain features in realtime, opening the door to simulate off-road vehicle mobility on different terrain and soil types. Tracked vehicles were initially studied since they are more suited for off-road travel. The modeling issues pertaining to tracked vehicle dynamics, soil mechanics and mobility are discussed in this document.

23.1.2 Definitions

vehicle dynamics model - A model that describes the motion and articulation of a simulated vehicle. These models can range from simple kinematic models to complex kinetics where many forces are used to determine resultant motion. Control systems can also be brought into play so that the simulated response better mimics the real world vehicle.

soil dynamics model - A model that describes the "motion of" (*i.e.*, the changes to) various attributes of the soil based on the characteristics of the soil. Probably the best example of this type of model has to do with soil slumping. In the absence of external restraint, soil in an unstable configuration will fall, or slump (36). These models can span a spectrum from simple heuristics to complex force calculations. In addition, these models can be further segregated into surface and volumetric types.

surface soil dynamics model - A soil dynamics model whose notion of the soil is defined by the boundary between the ground and the atmosphere. For any location there is exactly one value; in other words, there is no stratification. Many soil attributes are surface attributes. The most well known surface attribute of soil is elevation.

volumetric soil dynamics model - A soil dynamics model whose notion of the soil considers the attributes of the soil volumetrically. For any location there can be many values, depending on the depth of the sample and the degree of stratification. Certain soil attributes make more sense from a volumetric standpoint. An example relevant to mobility is soil strength.

mobility model - A model that concerns itself with effects on a vehicle caused by the interaction of the vehicle with the terrain. Mobility models are concerned with such calculations as tractive force and slip. Simple mobility models might use these calculations to modulate vehicle performance. More elaborate models might calculate forces that become inputs to the vehicle dynamics model.

trafficability model - A model that concerns itself with effects on the terrain caused by the interaction of a vehicle with the terrain. Trafficability models are concerned with calculations such as how the soil strength changes as vehicles drive over a particular path. This model is used interchangeably with mobility model by some authors.

23.1.3 Tracked Vehicles on Dynamic Terrain

The armored vehicle launch bridge (AVLB), a bulldozer and the mine clearing breacher (GRIZZLY) were modeled to drive on dynamic terrain. The mobility work on AVLB is discussed in (27). The trade-offs of using theoretical and empirical mobility models are also discussed in that document. The principles involved in modeling the vehicle dynamics and mobility remain the same for tracked vehicles used in DIS, viz. AVLB, GRIZZLY, HAB, M1A1.

The rut-depth models used for the breacher could be modified to represent a typical bulldozing operation. The steering and transmission of most bulldozers are different from the ones discussed in this document. This document is confined to the breacher simulation. The following parameters were the broad functional requirements for the simulation of the breacher and were representative of the capabilities of the prototype vehicle. These were coupled with the man-in-the-loop requirements of the simulation. The database requirements are discussed in "Soil Attributes" on page 289.

23.1.4 Functional Requirements

- Ploughing
- Automatic Depth Control System
- Ford Crossing
- Slope Climbing
- Mobility
- Man-in-the-loop requirements

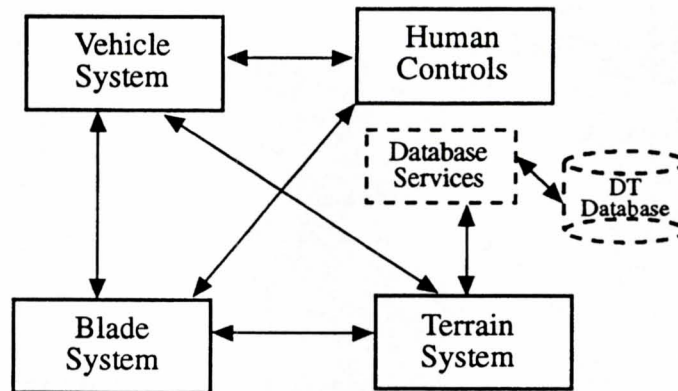
23.1.5 Mathematical Modeling Requirements

The objective is to obtain an accurate representation of the behavior of the system simulating the functional requirements. A deterministic, lumped parameter, constant coefficient, continuous time approach is used to develop the model. The breacher is a vehicle with complex interactions between the sensors, terrain, blade kinematics, blade controls, vehicle dynamics, and the driver. It is of great concern what subsystems must be represented and with what relative accuracies to simulate the functional requirements and also design

the database requirements. In addition, consideration must be given to realtime computational and visual requirements. A mathematical testbed (51) was constructed to prototype the system and study the couplings and the effect of interactions to decide the development of subsystem modeling. Since the state information of tracked vehicle is important in the simulation, the equations of motion are confined to the vehicle system.

23.2 Simulation Flow

Figure 113 is the proposed simulation flow representative of the coupling and interaction of various systems. The system decomposition, description and assumption are as follows.



0703-6976

Figure 113. Simulation Flow

23.2.1 Blade System

23.2.1.1 Terrain Mapping System (TMS)

The TMS uses three microwave radar arrays mounted on the blade that looks ahead of the left, center and right tracks. The points obtained are mapped to describe the surface ahead. This function is emulated by querying the database. The dynamic terrain database service is queried with a rectangular patch to get the elevation map. The look ahead is dependent on the projected speed, the time step used for the simulation cycle and the current position of the blade tip. Certain locations on the patch that conform to the loci of the blade tines are used as the reference heights for the blade ADC system.

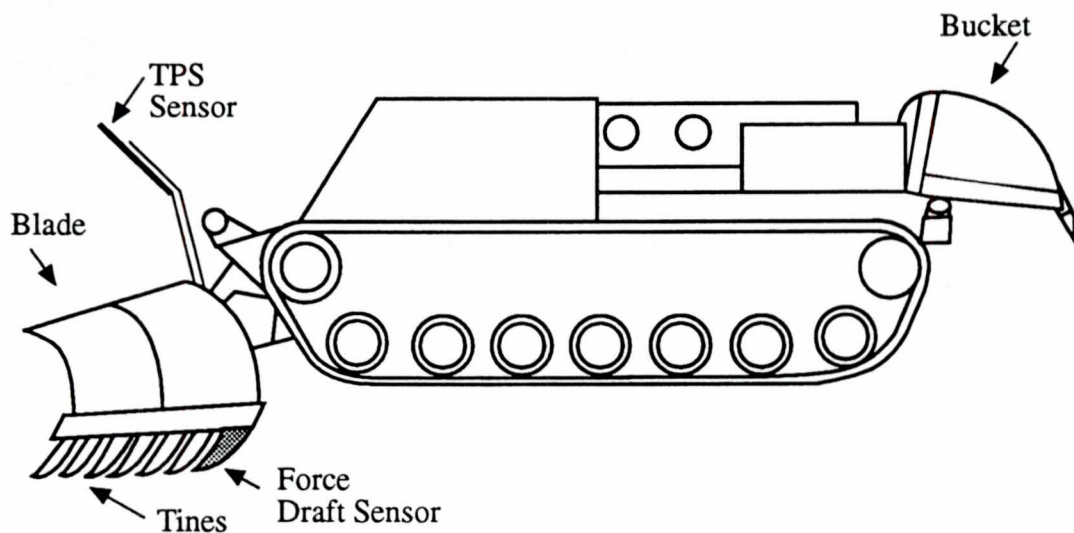


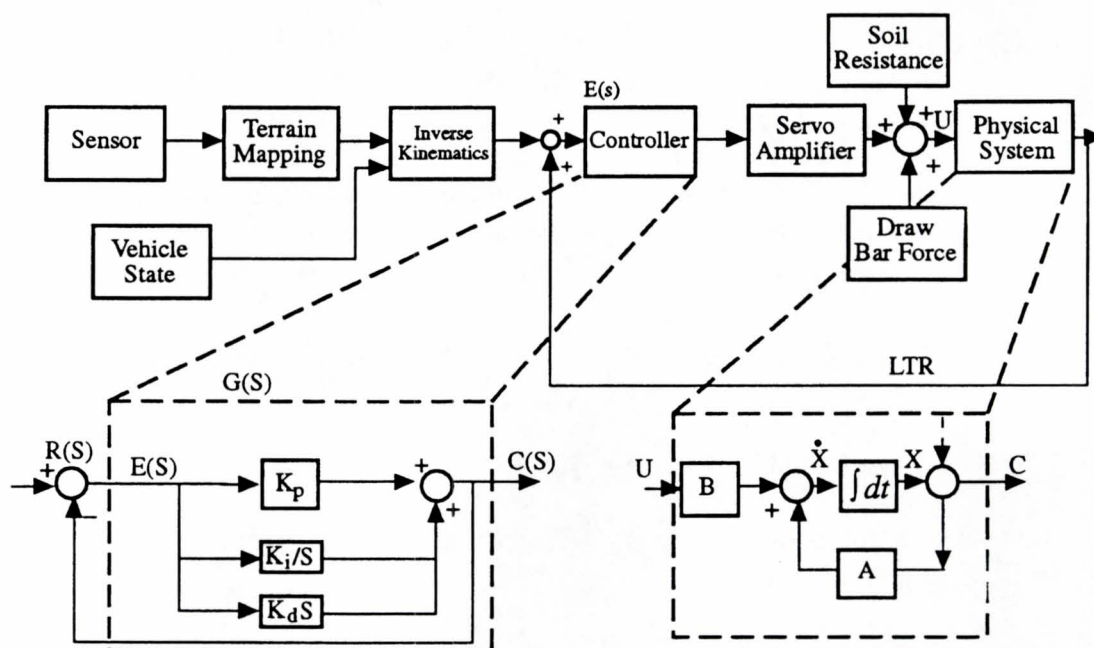
Figure 114. Schematics of the Breacher

In the actual vehicle the dug depth is determined using the force draft sensor. This is simulated by querying the database at locations just under the blade, after running the rut-depth model and soil piling models.

23.2.1.2 Automatic Depth Control System (ADC)

The ADC system maintains the mine clearing blade at a specified depth under the soil to clear mines aside. The hydraulic cylinder provides the required lift or digging depth to the blade. The changes in the blade depth beneath the soil, due to the terrain elevation, sinkage, or the orientation of the vehicle are compensated by the change in length of the hydraulic cylinder. The control system assists in driving the servo cylinder. Figure 115

shows the TPS and blade ADC flow in the simulation.



0703-6874

Figure 115. TPS Control Flow

There can be many control combinations for setting the reference height mapped by the radar. In the current simulation the center tip of the blade is assumed to be the reference height. Additional information on the actual hardware implementation is required to make additional judgements. The error signal is the difference between the reference height set by the sensor and the current blade tip elevation. This error signal drives the controller and hence the hydraulic cylinder. Depending upon the constraints of the mechanism, the resistance to cutting, velocity of the vehicle, changes in the attitude of the vehicle, response of the blade cylinder *etc.*, the blade acquires a new position. This position is checked against the reference and corrected accordingly. The frequency of the blade control loop could be different from the other integration cycles.

23.2.1.3 Blade Mechanics

There was not enough information available on the blade structure to add inertia force analysis in the simulation cycle. Hence the lag in the blade system is ignored relative to the simulation time step.

23.2.2 Vehicle System

The vehicle module is comprised of an engine, torqueconverter and a transmission. Performance curves for the AGT-1500 engine were used in the engine block. Criteria to throttle, the coupling between the torqueconverter and the load *etc.* are discussed in the design methodologies.

23.2.3 Terra-Mechanics

The motion resistance and the amount of traction available are represented in this model. Both of these affect the vehicle's mobility. The motion resistance is comprised of rolling motion resistance, sliding resistance and resistance to ploughing. The resistance due to the mine drag is not included in this study. Some of the information to calculate the above are derived from the NATO reference mobility model (73) and Waterways Experiment Station (WES) (70) results. The WES blade model is used as the rut-depth model. In addition, volumetric models were used for the soil slipping and piling (36) in front of the blade.

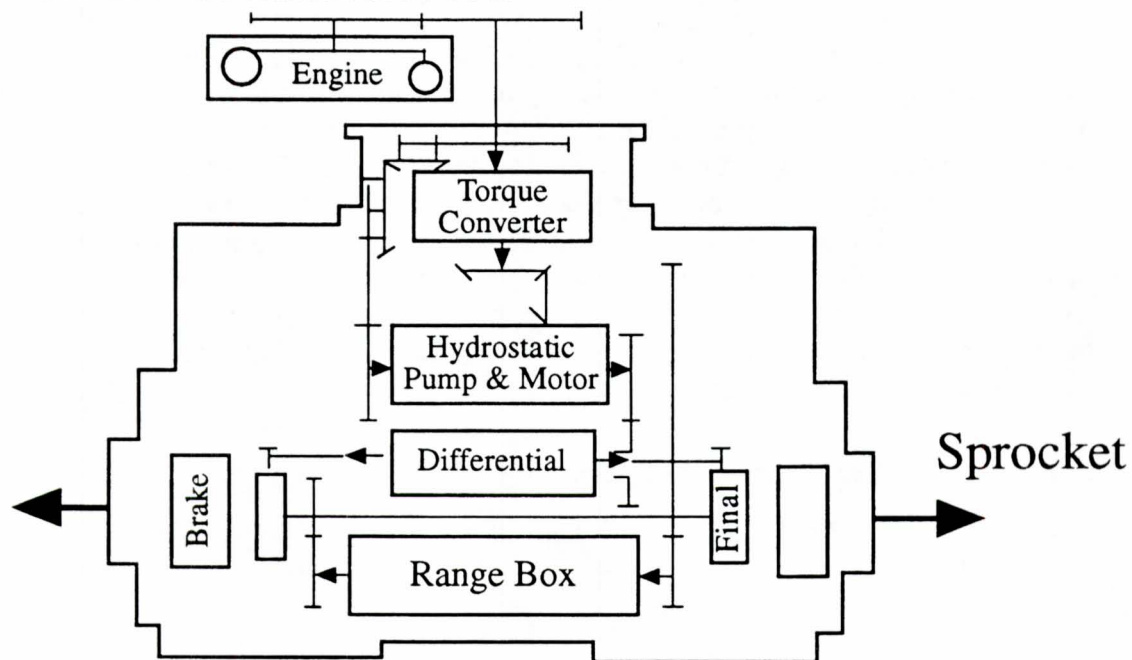
23.3 Design & Analysis Methodology

23.3.1 Vehicle System

The blade mechanism is built on a M1A1 chassis. Hence the engine and the powertrain are similar to that of M1A1.

23.3.2 Vehicle Dynamics

23.3.2.1 Simulation Flow



0703-6872

Figure 116. Vehicle System Schematics

Figure 116 illustrates the major vehicle subsystems and their coupling. The simulation flow follows the coupling chain. The modeling concepts of major subsystems are dis-

cussed below.

23.3.2.2 Engine

The breacher is powered by an AGT-1500 turbine engine. The steady state torque/HP-speed profile is used to determine various performance characteristics and operating points in the simulation of the engine. A three dimensional map in "Vehicle Performance Curve" on page 292 is used to obtain the HP for different engine speeds and throttle settings.

Some assumptions are made in finding the new engine operating point. Since these vehicles are designed for maximum traction, maximum torque is the criterion rather than fuel economy or power output considerations. The differential torque output for throttle changes are determined by keeping the engine speed steady and moving up or down the corresponding throttle curves.

23.3.2.3 Torque Converter

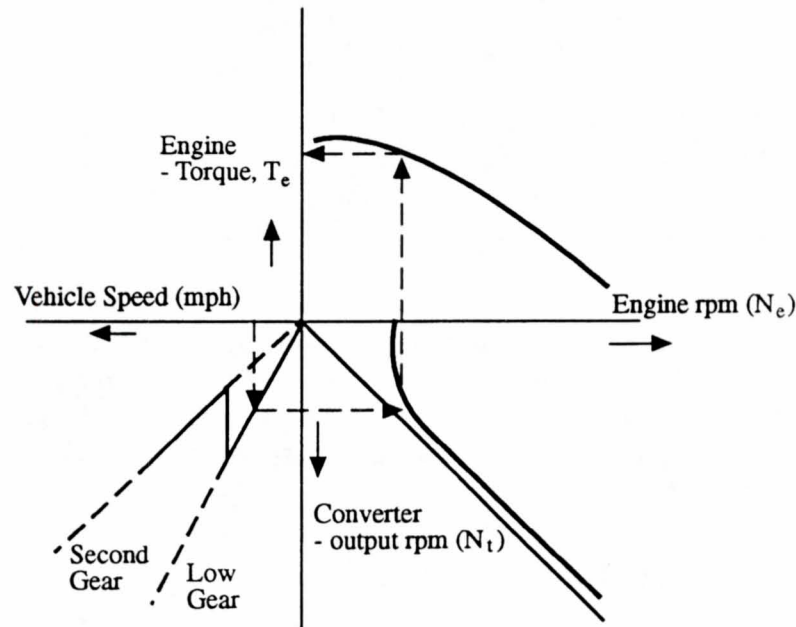
The engine is coupled with an automatic transmission, X1100-3B (64). The torque converter characteristics are modeled using the following relation (57)

$$\xi_T = 2.2 - 1.2\xi_S \quad (71)$$

where ξ_T is the Torque ratio, ξ_S is the speed ratio. Using the above relations, the following regression relation between the engine speed and torque converter output speed is established.

$$N_e = 1380.98 - 0.208704x + 0.0034437x^2 - 10^{-8} (3.61609) x^3 \quad (72)$$

where N_e is the engine speed and x is the torque converter speed in rpm. The engine idle speed is around 900 rpm. The above relations show a slightly higher stall speed due to non-availability of exact engine-torque converter characteristics. The following is the sequence of steps used to calculate the torqueconverter output due to load, throttle, and brake variations.



0703-6873

Figure 117. Power plant and torque converter characteristics

- Find current Vehicle Speed
- Find the current Gear ratio
- Find Torque Converter output rpm (N_t)
- Find last operating point of engine (N_e, T_e)
- Find Speed ratio

$$\zeta_s = \frac{N_t}{N_e} \quad (73)$$

- Find capacity factor i.e ability of the torque converter to absorb or transmit torque from K_t v/s ζ_s curve where

$$K_t = \frac{N_t}{\sqrt{T_t}} \quad (74)$$

- Find capacity factor for engine (assume $K_e = K_t$)
- Find new engine speed from K_e v/s N_e map
- Find new engine torque T_e from new engine speed and throttle settings
- Find new speed ratio(ζ_s)
- Find new output torque using Equation 71

23.3.2.4 Rangebox & Steering

An M1A1 is equipped with a turbine driven hydrostatically controlled differential. The range box provides four forward torque ratios (5.877, 3.021, 1.891, 1.278) and two reverse (-8.305, -2.353). In addition there is a final planetary drive providing a torque multiplica-

tion of 4.3. The sprocket radius is approximately 1.1 feet. There is an additional pivot steer to steer at stall. The ultimate requirement of the transmission is to meet sprocket torque and speed requirements for propulsion and steering. Steering of tracked vehicles is one of the most complex modules to model. *But the forces which must be brought into play, and the consequences of applying them, are less easy to understand, and it is to under-estimation of these forces that most of the troubles of tracklayer steering mechanisms can be traced* (43). The steering of the current tracked vehicles, unlike their wheeled counterparts, are force controlled under 40 mph. The simulation model is force driven and calls for an accurate representation of the track forces. Hence the use of conventional differential modeling to mimic track vehicle steering will only yield the track velocities and not track forces. The equations of motion and the mobility model depend on tractive forces. Elaborate modeling and testing were conducted to achieve reasonable steering control of the vehicle.

23.3.2.4.1 Skid Steering

The basic equation to calculate steer-track forces is given by the Merritt formula

$$F_o = K\mu \frac{W}{2} \quad (75)$$

where

F_o = longitudinal force required at outer track to overcome adhesion resistance to turning.

K = Proportionality factor, a function of length to width ration (60), μ = ground coefficient of adhesion, W = Weight of vehicle.

Equation 75 indicates that turning effort is independent of the velocity and radius of turn. It was difficult to initiate and control the steering with the above model. A slightly modified version of Equation 75 that handles rolling and sliding resistances across the track lengths for uniform pressure distribution is given by (76),

$$F_o = f_r \frac{W}{2} + \mu_t W \frac{l}{4B} \quad (76)$$

$$F_i = f_r \frac{W}{2} - \mu_t W \frac{l}{4B} \quad (77)$$

where

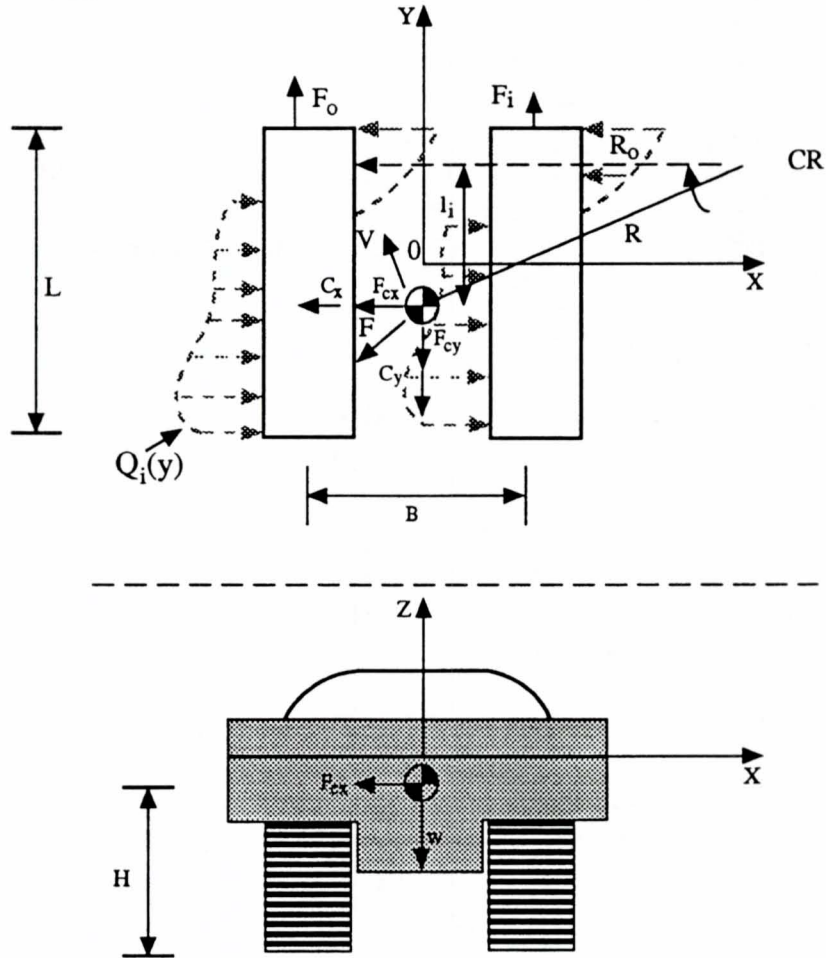
f_r is the coefficient of roll resistance, μ_t is the coefficient of sliding (lateral) resistance

The above equations indicate that in most practical cases of turning, the forces on the inner track are negative. This indicates the need for braking on the inner track. On implementing the above formulations the steering was very unsteady. Braking forces are not easy to estimate. This steering might be of help in a clutch brake steering employed in certain bulldozers. The use of the above steering principle leads to reverse steering going down a slope. The loss of forward thrust is also observed. This is a function of the soil type in addition to the geometry of the vehicle. Enough information could not be obtained on the transmission of M1A1 to determine when reversal of direction of inner tracks occur during a forward turn maneuver.

Therefore, additional power is required to initiate and maintain a turn while maintaining the forward velocity. The pressure distribution will always not be uniform and depends

upon the area in contact with the ground.

At higher velocities centrifugal forces affect the steering response. A complete analysis of the steering can be seen in (34). The following expression takes into account the effect of centrifugal forces.



0703-6871

Figure 118. Forces during a turn

$$F_{no} = \frac{W}{2} + C_x \frac{h}{B} \quad (78)$$

$$F_{ni} = \frac{W}{2} - C_x \frac{h}{B} \quad (79)$$

where F_{no} , F_{ni} are the normal forces on the outer and inner track respectively, C_x , C_y the component of centrifugal forces in the lateral and longitudinal directions,

$$C_y = M l_i \omega^2 \quad (80)$$

$$C_x = M R_o \omega^2 \quad (81)$$

where l_i is the distance between the cg and the pivot point of rotation along vehicle longitudinal axis and is given by

$$l_i = \frac{V_y}{\text{HeadingRate}} + cg_y \quad (82)$$

where cg_y is the center of gravity offset from geometric center along longitudinal axis, ω is the angular velocity of the vehicle about the center of turn, R_o is the component of radius of turn along the transverse direction.

The location of the instantaneous center of location has been treated differently by different authors (74)(6). Assuming uniform pressure distribution and pure rotation l_i can also be derived as (76),

$$l_i = l \frac{V}{2\mu_t g R_o} \quad (83)$$

In the steer analysis of the TVS (27) instantaneous centers of rotation were calculated for the points in contact with the soil. Accurate computation of track tensions were necessary to properly explain some of the terms. Nonetheless, the above approaches aid in the steady state force analysis. The radius of curvature of turn is an effect of the differential forces and slip of the tracks. The converse, *i.e.* use of the radius of curvature of turn (calculated using steering ratio or steer angle set by the driver interactively) as an input to calculate the required forces, will not always result in a desired steer response.

Equating the velocities we have,

$$R_o \omega = 0.5 [(V_o - V_{so}) + (V_i - V_{si})] \quad (84)$$

$$B \omega = [(V_o - V_{so}) - (V_i - V_{si})] \quad (85)$$

where V_o, V_i are the velocities of outer, inner tracks and are given by

$$V_o = R_{sprocket} \omega_o \quad (86)$$

$$V_i = R_{sprocket} \omega_i \quad (87)$$

V_{so}, V_{si} are the Slip Velocity of outer and inner track respectively.

Thus, using the above two equations we have,

$$R_o = \frac{B}{2} \epsilon \left[\frac{(1 - i_o) + (1 - i_i)}{(1 - i_o) - (1 - i_i)} \right] \quad (88)$$

where $\epsilon = \frac{\omega_o}{\omega_i}$ is the steering ratio. The track forces for outer and inner tracks are given by

$$F_o = F_{no} f_r + \frac{C_y}{2} + \frac{M_{rt}}{B} \quad (89)$$

$$F_i = F_{ni} f_r + \frac{C_y}{2} - \frac{M_{rt}}{B} \quad (90)$$

where M_{rt} is the resistive turning moment for a uniform pressure distribution and is given by,

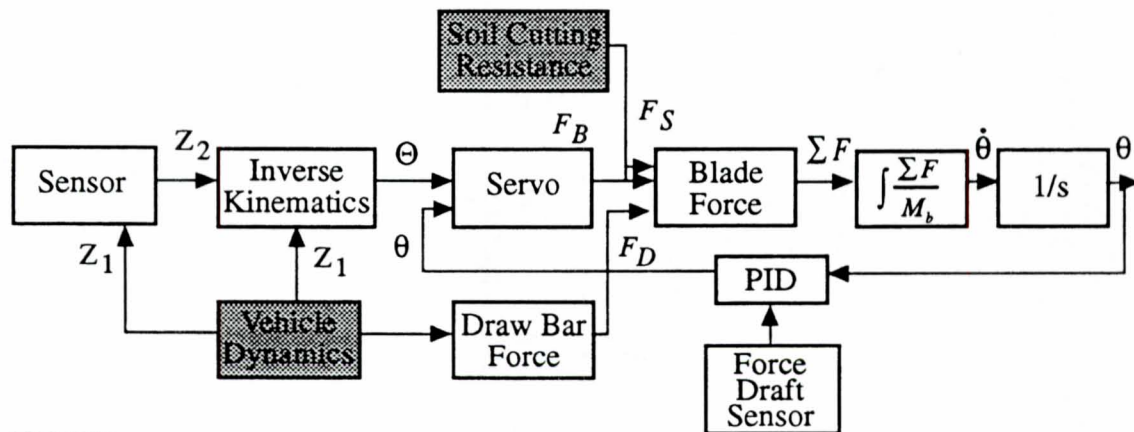
$$M_{ri} = \mu_i \frac{W}{2l} \left(\frac{l^2}{2} + 2l_i^2 \right) - \frac{WV^2 l_i}{gR_o} \quad (91)$$

Even though the above treatment of steering pertains to steady state turning behavior, it has been used to derive an estimate of track forces and power requirements. The transient force analysis of overcoming the slewing torque to achieve the desired steer response has been difficult. Literature indicates it is extremely difficult to measure and characterize the steer phenomena, either due to soil properties, the driver's difficulty to maintain a specified radius of turn at a specified speed, *etc.* The results of some of these studies are reported in <<TACOM c>>. The slewing force as a function of time during steer initiations <<TACOM b>> indicate an oscillatory behavior with a large amplitude of slewing force. *If the tracks are positively geared together to run with a given speed ratio, the radius of turn is determinate and the necessary forces are automatically brought into play; but the process is unfortunately not reversible and the application of those same forces without positive control of the track speeds does not produce the same radius of turn- or any turn at all <<Merritt, 1946>>.*

Field test results, accurate estimates of vehicle moments of inertia and driver inputs are required to further fine tune the transient response in a real time, interactive environment.

23.3.3 Blade Mechanics

23.3.3.0.1 Simulation Flow



0703-6870

Figure 119. Blade Control Simulation Flow

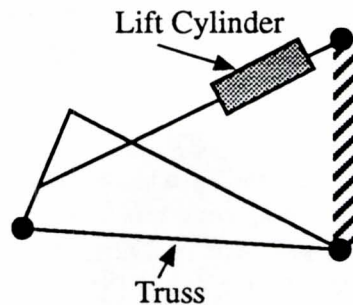
23.3.3.1 Blade Kinematics

The blade has a pitching degree of freedom to achieve the desired digging depth. The blade motion is controlled by the lift cylinder and two additional pitch cylinders, which correct the angle of attack of the blade tines. The two degrees of freedom are not independent of each other. The blade doesn't have a yaw degree of freedom, *i.e.* the pitch cylinders act simultaneously. To simulate the ADC system, an accurate representation of the

relation between the cylinder lengths, blade depth and blade angle is necessary. To simulate the blade trajectory, a mechanism design is essential.

23.3.3.1.1 Mechanism Design

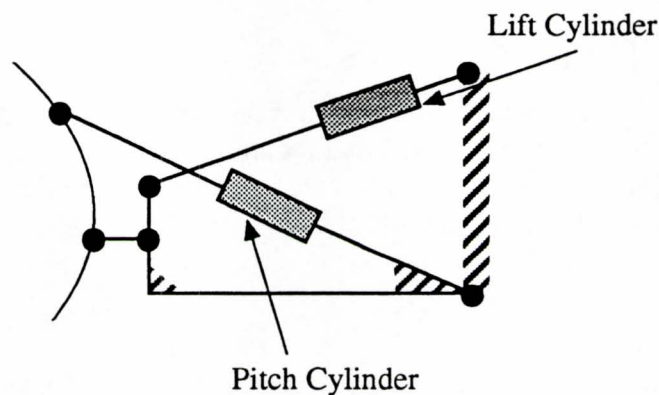
A planar mechanism has been chosen since it is adequate to represent the degrees of freedom of the blade. The spatial analysis can be extended by using the vehicle orientation to determine the location of blade tines along the blade width.



0703-6867

Figure 120. Blade Lift Mechanism

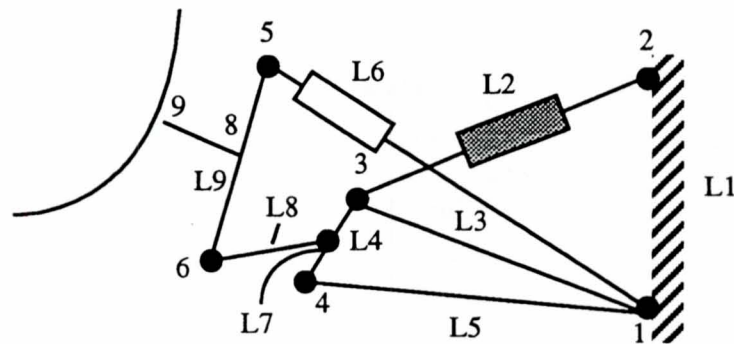
Figure 120 shows a simple planar mechanism that adequately represents a sturdy structure that provides lift to the truss structure. Since the orientation of the blade attached to the truss changes with the lift, additional links and pitch cylinders are provided to correct the blade inclination as shown in Figure 121.



0703-6868

Figure 121. Viable Mechanism Configuration

Figure 122 shows the various joints and link variables used in the simulation.



0703-6869

Figure 122. Simulated blade mechanism configuration

The coordinates of joint 3 are determined using the trajectory of link L3 and the change in length of the lift cylinder. The convex hull algorithms (52) is used to resolve the coordinates of joint 3. Similarly, the coordinates of joint 6 are determined.

23.3.3.1.2 Mechanism Simulation

The depth of dig or the lift cylinder extension is constrained by the length of the different links used in the mechanism. The prototype specifications only suggested a maximum depth of dig of 15 ± 2 inches. Therefore, the simulation had to be designed to obtain this depth by varying different link dimensions, consistent with the constraints that would still operate in the given work envelope.

23.3.3.2 Blade Controls

A simple linear feedback control system is employed to drive the actuator. The difference between the estimated reference depth (command input) and the current blade location (control output) determines the error for the position controller. The following proportional integral control law was sufficient to control the blade mechanism.

$$u(t) = k_p e(t) + k_i \int_0^t e(t) dt \quad (92)$$

where K_p and K_i are the proportional and integral gains. Since the simulation is goal oriented, the dynamic response of the blade system was not taken into account. The response time for the hydraulic system was assumed to be one fifth that of simulation time step integration.

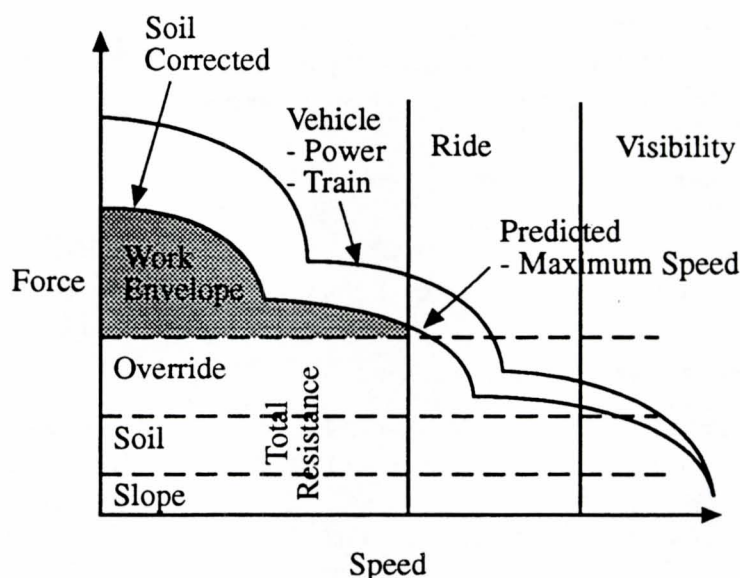
23.3.4 Soil Mechanics

23.3.4.1 Mobility

The capacity of the vehicle to move from one location to another on a specified terrain performing its primary mission. The blade and the bucket are mounted on the M1A1 hull.

The mobility of the breacher is similar to that of the M1A1 when the blade and bucket are stowed. The track-terrain interactions during the traversal of different soil types and during the ploughing operations are critical to vehicle mobility and the mobility of the other vehicles following the breacher. Mobility can be quantized with the speed made good on a specified terrain, or on mobility maps drawn for logistics purpose. Nato Reference Mobility Model (NRMM II) and the simplified mobility model (SAMM) are some of the mobility models used for this purpose.

23.3.4.1.1 NRMM II



0703-6866

Figure 123. Tractive force versus speed curve

The basic assumptions of the NRMM II model, viz. steady state, point to point travel, maximum power, makes the model unsuitable for interactive (man-in-the-loop) real time simulation. Empirical results are used to estimate the soil traction and motion resistances. The soil properties are derived as a function of Cone Index (CI) for different soil types. The basic formula for the drawbar force is given by

$$\text{Drawbar Force} = \text{Tractive Force} - \text{Motion Resistance} \quad (93)$$

The above formula is used to calculate the motion resistance and the maximum traction available. The following scheme is used to determine the maximum traction available and motion resistance for different slips on different soil types and soil strengths (CI).

23.3.4.1.2 Motion Resistance

$$\frac{R_p}{W} = A + \frac{B}{RCI + C} + D(RCI) \quad (94)$$

where A, B, C, D are constants depending on soil types (73), RCI is the excess soil strength for traction element (Cone Index - VCI), psi. The above relations are not inclu-

sive of the soil cutting resistance (72). The multipass degradation of soil is taken into account using multipass VCI (27).

23.3.4.1.3 Traction Available

The maximum available soil traction is determined using equation (93). The following empirical relations are used to determine the drawbar force at a specified slip, soil strength and soil type.

$$\frac{D_s}{W} = A_{ds} + \frac{B_{ds}}{s + C_{ds}} \quad (95)$$

where

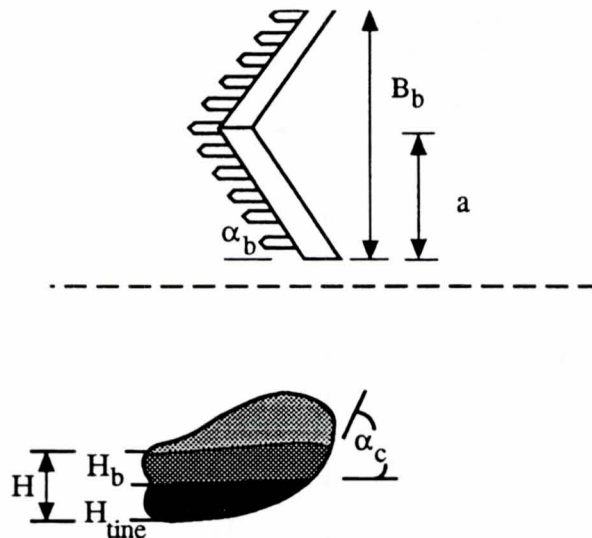
A_{ds} , B_{ds} , C_{ds} are constants <<WES g>> dependent on soil type, s is the slip of the track in percent, and D_s is the draw bar force available on maximum soil strength (300 RCI).

$$\frac{D_{snom}}{W} = A_{dn} + \frac{B_{dn}}{RCI + C_{dn}} \quad (96)$$

where A_{dn} , B_{dn} , C_{dn} are constants dependent on soil type and RCI the excess soil strength, D_{snom} is the maximum drawbar force at nominal slip.

23.3.4.1.4 Rut Depth Models

Ploughing operation is important to determine vehicle power requirements and for tracked vehicle mobility. The breacher has a ploughing capability of 15 inches. The top 6 inches of soil affect vehicle mobility. The soil properties are normally assumed constant for this depth. In the case of the breacher, additional data is required to interpolate soil properties between 6 and 15 inches. The new properties of soil after the plough effects the mobility of the breacher and also the vehicles following it.



0703-6865

Figure 124. Blade geometry

The WES ploughing model (equation (97) in "Soil Manipulation Procedure" on page 298) is chosen to represent the soil cutting and translating resistance needed by the breacher

model. The soil piling model (37) is used to pile soil alongside and in front of the blade. These formulations are more theoretical and use non-empirical soil properties. The shear force terms could be substituted with empirical values viz. function of CI. Details of cross conversion of some of these terms are dealt in the mobility document (27).

23.3.4.1.5 Soil Attributes

The database consists of any reasonable number of conceptual layers, each data layer corresponding to a particular attribute of the terrain representation (e.g. temperature, hardness, moisture content). A separate mathematical surface is used for each terrain database attribute in the Dynamic Terrain Database abstract datatype. Queries can be made at either single points or for an arbitrary four-sided region at any desired sampling density as shown in Figure 125. Therefore, for any point within the extents of the database, there exists a vector $(0.n)$ of information describing that point. This vector will contain elevation, as well as all other attributes specified for the particular environment. In addition to attribute values, surface normals and parameter values can be returned via variations of this query. The dynamic terrain database can support finer detail using less storage capacity than a polygonal mesh.

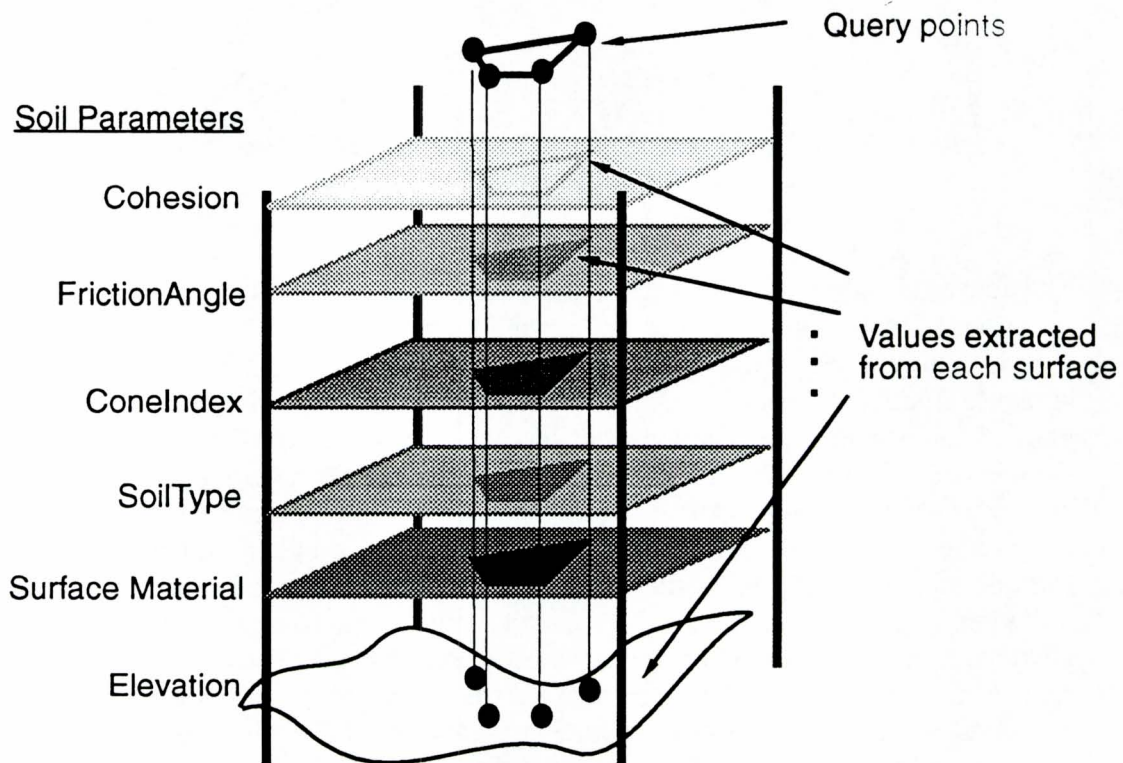


Figure 125. Soil Attribute Layer

Some of the commonly used soil parameters are discussed in "Soil Attributes" on page 293.

23.4 Conclusions

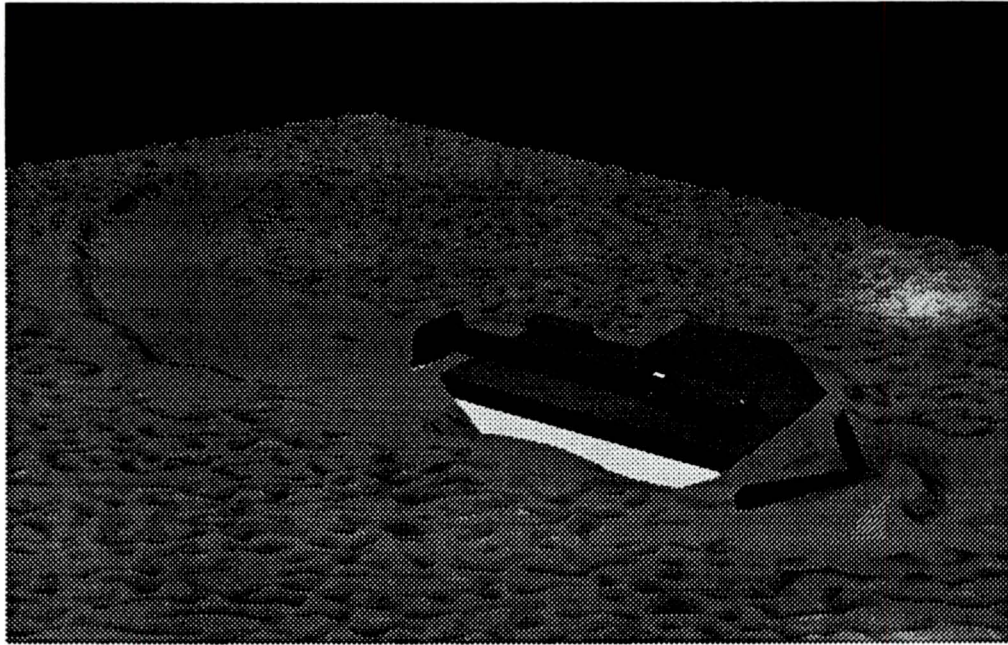


Figure 126. Breacher clearing a mine field

The above formulations were tested and implemented on a dynamic terrain testbed. The equations of motion were derived in the vehicle frame axis and integrated to obtain the state information. Figure 126 shows the breacher in operation and the cleared path. The breacher simulated most of the functionalities very well. The effects of mobility on fine grained and coarse grained soil were observable. The effect of coupling between various vehicle and soil subsystems, viz. blade control response time, database resolution, vehicle dynamics observed in the mathematical testbed, were also observed in the actual implementation. The vehicle velocity had a significant impact on the after cut of the soil. Uniform depth of cut along the blade face was observed only at vehicle speeds less than five miles per hour on fine grained soil. The vehicle specifications indicate 5.8 mile per hour as the maximum speed during ploughing, but does not indicate the soil type. The depth of cut was uniform when the time period for the blade hydraulic system was less than 0.05 sec. This indicates the necessity of a separate integration cycle for the blade when running at a different frequency. Since enough information was not available on blade inertias, the blade error correction cycle was run at a higher frequency proportionate to the simulation time step.

The ford crossing ability was constrained to crossing drybeds. The mean absorbed power by the human body also limits the speed. This is related to the surface roughness. The lack

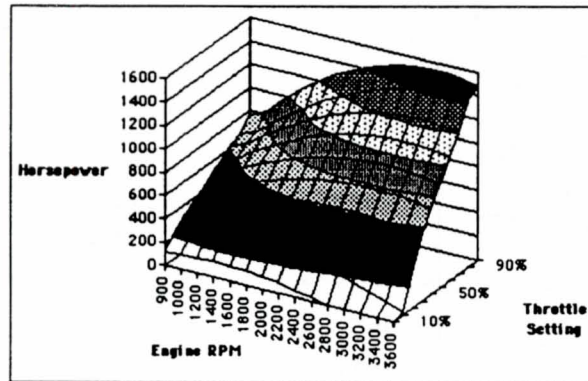
of methodology to compute the surface roughness after the ploughing limited the implementation of surface roughness model. The output from the vehicle dynamics (WES -VDYN) preprocessor indicates the lower speed limit for a 6 watt absorption power on a 5.5 rms surface is around 20 m.p.h. on the M1A2. Since these limits were higher than the breaching speed, the effect of surface roughness was considered negligible.

Measures of performance are at once extremely hard to define in such closed-loop continuous system simulations because of the tight coupling between components, and yet are essential to the development of the models. *e.g.* The rut depth achieved and the distance traversed on a one foot resolution soil and a four foot resolution soil are different for the same vehicle operating conditions.

Successful simulation of such systems calls for the identification of all interactions that have bearing on the intended use of the model, as well as how these interactions are measured and how much variance could be tolerated. Performance measures are needed for all components of the simulation, including the tracked vehicle, the terrain representation, database capacity to model the rapid changes in attributes, the hardware used for implementation, and of course, realtime performance. Experiments could provide insight into first order interactions if test data was available for various combinations of parameters of interest. System identification techniques could then be used based on the experimental results. Parametric analysis could be used to identify the sensitive interactions.

The current implementation not only serves the purpose of training simulator, but also in design and testing of vehicle subsystems on varying terrain and driving conditions.

23.5 Vehicle Performance Curve



23.6 Soil Attributes

Table 1: Dry Soil Attribute Values* Used in the Database

USCS Soil Type	No	Soil Description	Friction Angle deg	Dry Density pcf	Swell Factor	Cohesion psi (ULimit)	Cone Index (0-6") WI=1
SW	1	Well Graded Sands, gravelly sands, little or no fines	43	93.6	1.05	0	300
SP	2	Poorly graded sands or gravelly sands, little or no fines	34	93.6	1.05	.35	300
SM	3	Silty sands and sand-silt mixtures, nonplastic fines	36	93.7	1.10	.7	122
SC	4	Clayey sands & sand-clay mixtures, plastic fines	36	97.4	1.10	1.4	101
ML	7	Inorganic silts and clayey silts of low plasticity	32	73.7	1.20	1.75	60
CL	6	Inorganic clays of low to medium plasticity, lean clays	30	86.8	1.30	3.15	85
OL	11	Organic silts and organic silty clays of low plasticity		77.4	1.35	2.1	103
MH	10	Inorganic silts, micaceous, or diatomaceous silty soils	28	66.2	1.20	2.45	61
CH	9	Inorganic clays of high plasticity, fat clays	22	85.5	1.45	3.5	135
OH	12	Organic silts and clays of medium to high plasticity		52.5	1.45	2.8	115
PT	20	Peat and other highly organic soils	-		1.45	1.05	190

*The values are a set of possible mean values compiled from (67), (71), and (72). In general these parameters are a function of other conditions viz. relative density, moisture and a range for the above values are normally suggested.

USCS Soil Type	No	Soil Description	Friction Angle deg	Dry Density tcm	Swell Factor	Cohesion tsm (ULimit)	Cone Index (0-6") WI=1
SW	1	Well Graded Sands, gravelly sands, little or no fines	43	1.4993	1.05	0	300
SP	2	Poorly graded sands or gravelly sands, little or no fines	34	1.4993	1.05	.037	300
SM	3	Silty sands and sand-silt mixtures, nonplastic fines	36	1.5	1.10	.0745	122
SC	4	Clayey sands & sand-clay mixtures, plastic fines	36	1.56	1.10	0.14907	101
ML	7	Inorganic silts and clayey silts of low plasticity	32	1.18	1.20	0.18634	60
CL	6	Inorganic clays of low to medium plasticity, lean clays	30	1.39	1.30	0.3354	85
OL	11	Organic silts and organic silty clays of low plasticity		1.239	1.35	0.2236	103
MH	10	Inorganic silts, micaceous, or diatomaceous silty soils	28	1.06	1.20	0.2608	61
CH	9	Inorganic clays of high plasticity, fat clays	22	1.369	1.45	0.37268	135
OH	12	Organic silts and clays of medium to high plasticity		0.84	1.45	0.298144	115
PT	20	Peat and other highly organic soils	-		1.45	0.1118	190

24.0 Digging Tool Model

24.1 Introduction to the Digging Tool Model

24.1.1 Background

The interaction between the movement of a rigid object on a terrain surface is important for a Dynamic Terrain (DT) simulation. A moving object such as a bulldozer blade, the front-end of a mine breacher, or a rock can change the underlying terrain. These moving objects all involve similar digging stages; however, the geometry of the digging face differs. The ability to change the tool geometry and visualize its effects on soil manipulation can be a useful tool to designers of various vehicles or digging tools such as the breacher blade.

It is not the intent of this model to cover every possible digging tool, but to provide a digging function based on a hybrid of both physical and kinematic models (see Section 6.1.1 on page 81 for a description of these models). The implementation of this model is referred to as the Digging-ToolModel.

24.1.2 Problem Statement

The general digging tool should support some basic functions encountered in a typical DT simulation. It will supply firstly the capacity to model a soil cutting and manipulation tool of different geometrical shapes, secondly, the capacity to model soil mechanics for different soil and terrain attributes and thirdly, the capacity to visualize the changes interactively. The model should provide a mechanism for soil plowing that is easily used by the simulation designer. The designer should be concerned simply with the geometric description of a digging object while the model shelters it from the soil mechanics of a digging operation.

24.1.3 Solution

The digging tool model allows the design and testing of a soil cutting tool in a DT simulation without requiring the tool designer to have much knowledge of soil dynamics. The tool's digging face is determined according to its orientation with respect to the terrain. In the current implementation, the tool designer supplies a geometric representation of the blade, or tool surface, to the DiggingToolModel. It extracts the geometric information of the tool face and provides methods to allow soil excavation.

24.1.4 Constraints/Assumptions

The implementation includes representations of both the moving rigid object and the soil. This implementation gives the most realistic model possible with the efficiency demanded by a real-time simulation. Thus, there is a trade-off between a purely physical and a purely geometric model. For the proposed design the following assumptions have been made:

- The model is compatible with Terrain Services. Terrain elevations are modified according to the digging algorithm, then passed back for re-merging into the global terrain.
- The soil/object interaction consists of digging, piling, and slipping. The digging and piling of the soil are accomplished while conserving the soil's volume. A soil slippage model has been implemented for DT as the Soil DTR and is used to determine if a portion of soil will slide. Therefore, this implementation uses a kinematic model for the digging and piling stages and a physical model for the soil slippage stage.
- The tool's displacement of the soil will contribute only to neighboring elevation posts and will pile to a height limited by the height of the tool.
- Movement of the soil will not occur during placement and retraction of the tool from the soil; thus, soil displacement occurs only under the tool.
- The tool's displacement of the soil does not allow soil to fall behind the direction of movement; thus, soil will be placed in front or along the sides of the object -- not behind it.

24.2 Analysis

In this section, the digging procedure is generalized in an effort to allow any digging tool surface to manipulate the terrain. A methodology to consider different shapes of the tool is described in the following section.

24.2.1 Digging Tool Geometry

The outer surface of the digging tool is represented by a rectangular mesh. Only the surface of the tool is considered for interaction with the soil. Attention is given to the tool surface facing the direction of motion. The current model allows only forward motion of the tool and considers 1 tool surface; however, the same algorithm could be extended to use any tool surface in the direction of movement.

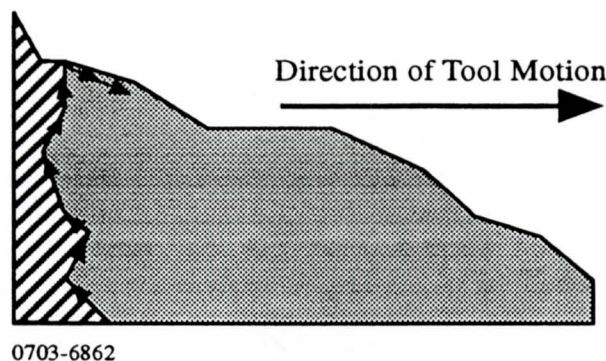


Figure 127. Soil motion along digging tool

The designer assigns the shape of the tool; however, the digging model may reshape the tool's mesh size to match the ratio of terrain surface size to the tool size. In general, the tool facets

should be smaller than the terrain cells. If the tool facets are smaller than the terrain cells, a tool facet will interact with at most with two terrain cells. In actual implementation, the ratio is determined by the digging tool configuration file's scaling parameter and the resolution of the terrain. Variations in digging depth and terrain resolution change the number of tool facets interacting with the soil. This is determined dynamically by the digging model and effects both volume conservation and piling processes. The increment in the cutting depth will increase the volume displaced. The resolution change will only affect the precision of the volume conservation since the integration formula uses finer cells (Section 24.2.2.2: Piling Stage).

24.2.2 Soil Manipulation Procedure

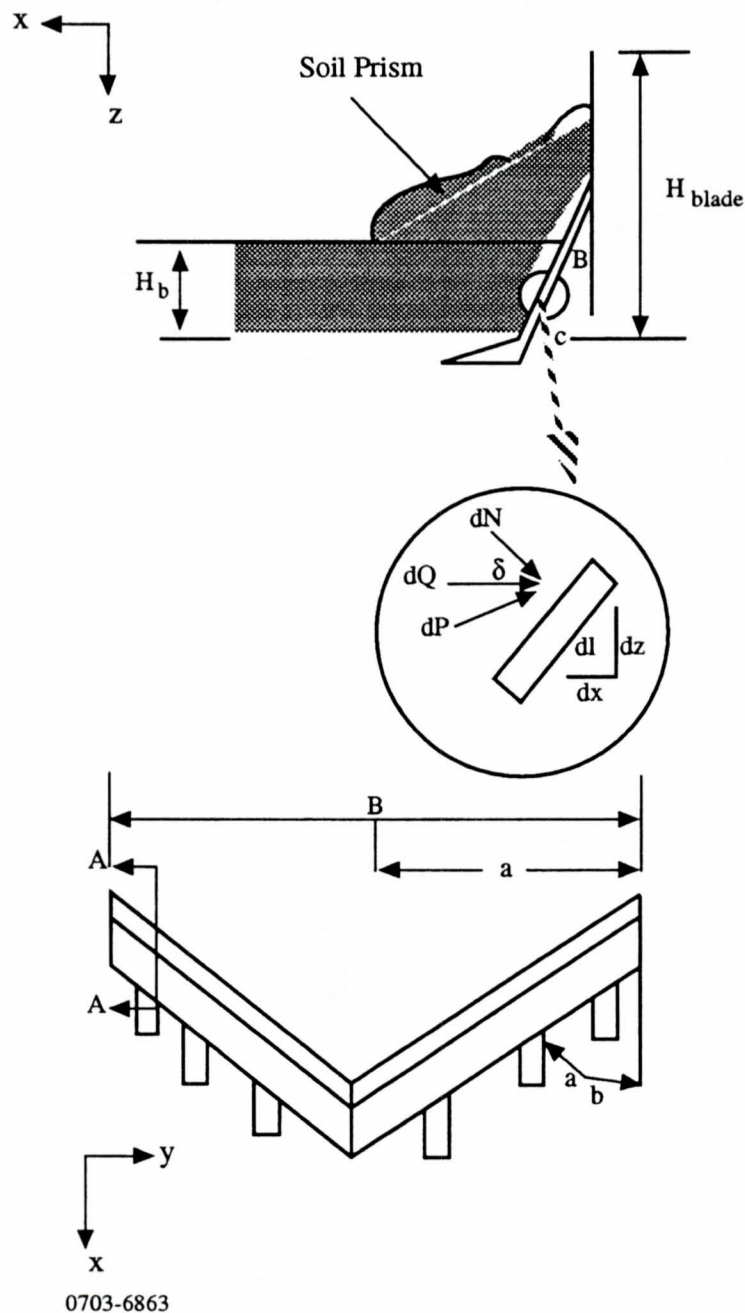


Figure 128. WES Blading Model

Excavation of soil by a tool consists of three processes: cutting, piling and slipping. Before discussion of these stages, the soil resistance force is detailed. This resistive force is central to the digging procedure.

The U.S. Army's Waterways Experiment Station (WES) blading model was selected for implementation since current Dynamic Terrain digging applications involve bulldozing and breaching.

The WES model was developed for the Combat Mobility Vehicle and assumes constant depth cutting and a soil that behaves as a rigid-plastic frictional material¹. This simplified model is based on the theory developed by Farr and Baladi (1988) and Balovnev (1963). (The figure on the previous page depicts the variables used in the following analysis.) The equation for the horizontal plowing, or resistive cutting force, follows:

$$f_r = 2H_b A \left[\frac{\gamma H_b}{2} + c \cot \phi + \gamma_p H_{pr} + \gamma H_b + (\tan \delta_p + \tan \phi_p) (1 - \sin \phi_p) \frac{\gamma_p}{2} H_{pr}^2 \right] \\ \left[a \left(1 + \frac{\tan \delta}{\sin \alpha_b} \sqrt{\frac{1}{\sin^2 \alpha_c} - \sin^2 \alpha_b} \right) \right] + \gamma_p H_{pr}^2 \frac{a}{\tan \phi_R} + \cos \delta_p \quad (97)$$

where A = passive earth pressure constant

$$A = \frac{\cos \delta \left(\cos \delta + \sqrt{\sin^2 \phi - \sin^2 \delta} \right)}{(1 - \sin \phi)} e^{\left(2\alpha_c - \pi + \delta + \sin^{-1} \frac{\sin \delta}{\sin \phi} \tan \phi \right)} \quad (98)$$

c = soil cohesion,

α_c = cutting angle of plow blade,

α_b = angle between blade and forward motion,

H_b = depth of plowing from terrain surface,

H_{blade} = height of blade,

H_{pr} = height of soil prism,

B = angle between the resultant and horizontal component of cutting force,

γ = density (i.e. total unit weight of the soil),

δ = friction angle between soil and plow face,

ϕ_p = internal angle of friction for the soil prism,

ϕ_R = angle of repose,

and

ϕ = angle of internal friction.

In actual implementation, some parameters are set to the default values to simplify the simulation such as the angle of repose. This is set to 45 degrees. The cutting angle of the plow blade is set to 90 degrees and the angle between the blade and forward motion is set to 90 degrees. The other parameters can be changed during the simulation.

24.2.2.1 Digging Stage

The digging stage consists of placing the tool in the soil, moving the soil, and retracting the tool from the soil. Placement and retraction of the tool is complicated because of the angle between the digging direction and the terrain surface; thus, it is assumed that movement of the soil will not occur during placement and retraction of the tool from the soil. The initial placing of digging tool

1. John Farr, Conrad Rabalais, Robert Underwood III, and Richard Ahlvin, "Mobility and Plowing Capabilities of the Combat Mobility Vehicle", US Army WES Geotechnical Laboratory, April 1991.

will have to change the pitch angle and digging depth all the time so it is difficult to accommodate all the effects in the model. Position and depth of the tool with respect to the terrain are supplied as parameters to the digging model. The tool's motion is considered once the tool is placed into the soil and moved along the terrain.

Since only forward motion of the tool is considered for soil displacement, an easy way to accommodate reverse movement of the tool involves simply moving the tool. The reverse path is the inverse of the forward path; thus, there is no digging effect.

A simple dynamic model correlates the range of cells affected by volume displacement with the tool's acceleration. The distance affected by the horizontal force is determined by the mass driving the digging tool, the acceleration, and physical properties of soil where

$$\Delta d = v\Delta t + \frac{f_{cut} - f_{hr}}{m}\Delta t^2 \quad (99)$$

where

f_{hr} = soil resistance in horizontal direction,

f_{cut} = the cutting force,

m = mass of the vehicle and digging tool,

Δt = time interval since previous update,

and

v = current velocity.

The horizontal resistive force, f_{hr} , is obtained from the WES blading model (See Equation 97).

The total soil volume effected by the acceleration is

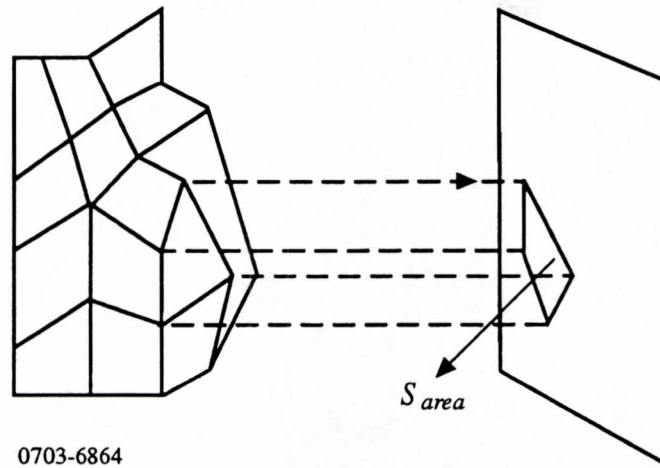
$$\Delta V = \int_{\Omega} \Delta d \Delta s \quad (100)$$

where

Δs = projected area of soil displacement.

Once the distance travelled has been determined, the volume of displaced soil is computed. Only the facets of the tool under the terrain surface and facing the direction of motion are considered in this calculation. A method to accomplish the numerical integration for Equation 100 is to project the facets of the digging tool onto a plane, or several planes, relative to the vehicle. The surface of the digging tool is projected onto a vertical plane in the direction of motion as shown in the figure

below. Each facet is projected as a small polygon with area S_{area} .



0703-6864

Figure 129. Projection of Digging Tool Facet onto a Plane

The total volume, ΔV , displaced by tool surface facets in the time interval Δt is approximated by the summation of volume contributed by individual facets.

$$\Delta V = \sum_{i=0}^m \sum_{j=0}^n S_{area_{i,j}} \Delta d \quad (101)$$

where

m = number of cells in x direction,

n = number of cells in y direction.

and

S_{area} = the area of a facet cell on the digging tool face.

A simple 2-D clipping algorithm is used to deal with facets having only a portion under the terrain surface. The clipping procedure simply apply removes any facets or portions of facets from above the terrain surface. If all facets are under the terrain surface, no clipping is needed.

24.2.2.2 Piling Stage

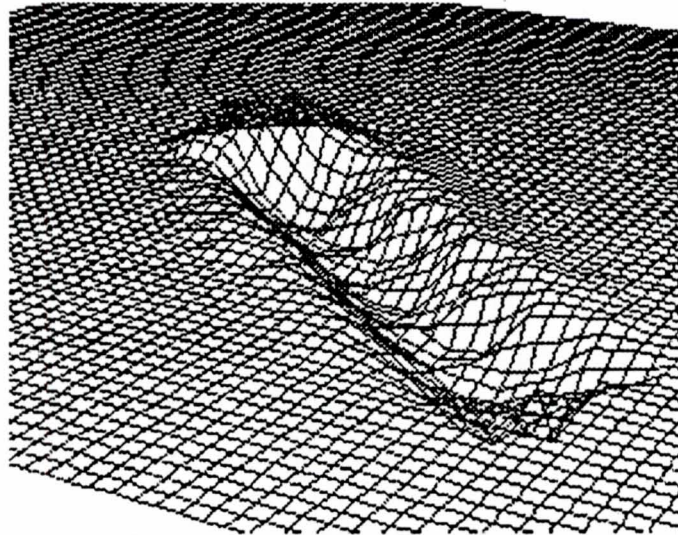


Figure 130. Front and side piling of soil

This implementation uses a geometric model for piling soil in front and along sides of the tool as shown in the figure above. When soil is excavated, it undergoes expansion. This increase in volume is computed using the soil's swell factor, σ , which is described in "Soil Attributes" on page 293. The excavated volume is

$$\Delta V_{excavated} = \sigma \Delta V \quad (102)$$

The final step in the digging stage involves placement of the excavated soil volume. Determination of the effected neighboring cells is a function of tool and terrain parameters

$$\text{Cells}_{affected} = f(m, v, f_{cut}, f_{hr}) \quad (103)$$

24.2.2.3 Soil Slippage Stage

The Soil DTR (Section 6.1 on page 81) is used to slump the soil after piling. This algorithm uses soil properties to determine the stability of a given soil configuration by calculating the critical angle above which sliding occurs and the force which pushes the soil mass along the failure plane.

24.3 Implementation

24.3.1 Problem Statement

The digging tool model uses the Dynamic Terrain Data Base (DTDB) or Terrain Services for terrain patch elevation and soil property information. These values are used in all stages of the excavation process. Digging tool geometric data is provided by the modeling data file described in Section 24.3.2.3: Digging Tool Geometry File. After initialization, the application excavates soil according to the digging, piling and slumping procedures explained in Section 24.2.2: Soil Manipulation Procedure when the tool is beneath the terrain surface. The modified soil patch is returned to the DTDB for integration into the terrain database.

24.3.2 User's Guide

24.3.2.1 A typical program structure.

```
//Instantiate Digging Tool DTR
digtool= new DiggingToolModel("digtool.cfg");

//Instantiate Soil DTR
soil = new SoilModel(debug);

//Instantiate Dynamic Terrain Database
dtdb = new DynamicTerrainDatabase("test digtool", filename);

while (quit_demo != TRUE)
{
// Fill the dynamic and geometry structure for digging. Those parameters include
// the velocity, vehicle mass, the digging depth and etc. Check the data structure for
// the details.

    digtool->initDigging(Tool location, tool orientation, &pul, &pur, &plr, &pll, &num_in_s,
        &num_in_t, DynamicPara *dynamic, GeometryPara *geometry),

//Query DTDB for elevation values
    dtdb->getAttributeValuesAndNormals(0,
        demo.pul,
        demo.pur,
        demo.plr,
        demo.pll,
        demo.pnum_s,
        demo.pnum_t,
        demo.old_h,
        demo.old_n );

//Determine soil volume and location to be excavated
    digtool->digging(new_depth, demo.old_h, demo.new_h);

//Relocate newly excavated soil
    digtool->piling(new_depth, demo.new_h, swell_factor);

//Invoke soil slippage algorithm
    soil->slumpSoil(10, demo.pnum_s, demo.pnum_t, demo.new_h);

//Return modified data to DTDB
    dtdb->handleUpdate(0,
```

```

demo.pul,
demo.pur,
demo.plr,
demo.pll,
demo.pnum_s,
demo.pnum_t,
demo.new_h );
}

```

24.3.2.2 Configuration File

The digging model requires initialization of several tool parameters prior to a simulation. Below is a sample configuration file with comments. The syntax is consistent with that of Terrain Services' configuration file. Lines beginning with "#" are comments and blank lines are ignored. Most key words are followed by a value on the same line while some key words are followed by a number indicating how many lines belong to this attribute. The header string "DIG_TOOL_CONFIG" must appear on the beginning of the file or the file is invalid.

```

DIG_TOOL_CONFIG
# The file tag: DIG_TOOL_CONFIG
# The physical features of the digging tool

# The data file name for digging tool wireframe

# The breacher blade
data_file ./flt/make.mesh

#The orientation of the digging plane that specify which plane is used for digging
#We should know which plan is used for digging since digging tool shape 3-dimensional.
# XY=1 YZ=2 XZ=3 Should be XZ or YZ plane?
digging_plane      3

#The reference point of the digging tool used for the geometric transformation.The 3D geometric
# transformation a reference point to evaluate the transformation matrix. Typically, the reference
#point is the low-left coner of the digging tool model. (These values can be used for most cases.)

ref_point_x  0.0
ref_point_y  0.0
ref_point_z  0.0

#Initial digging direction vector for control the movement of digging tool
ddv_init      0.0

#The digging step size(for testing only)
digging_step 100.0
#The piling step size (for testing only)
piling_step 330.0

#The initial scaling for the digging tool in order to match the terrain coordinate
scale_x  50.0
scale_y  50.0

```

scale_z 50.0

#The initial rotation to turn the correct digging plane.

rotate_x 0.0

rotate_y 0.0

rotate_z 0.0

#Initial digging location: for the translation

loc_x 1000.0

loc_y 1000.0

loc_z 200.0

#The corner of terrain patch for digging

upleft_x -3.0

upleft_y 11.0

upright_x 12.0

upright_y 11.0

lowright_x 12.0

lowright_y 0.0

lowleft_x -3.0

lowleft_y 0.0

#The digging tool geometry description follows. This was added for ease in integration with the bulldozer simulator; therefore, the mesh file is ignored.

stretch_count 3

2.0 0.0

7.0 2.0

12.0 0.0

All values except for the mesh file name, digging plane orientation, reference point, terrain patch corners, stretch count, and stretch values are ignored in an integrated simulation such as the use of the digging tool for the blade of the bulldozer. However, these are necessary for a stand-alone demonstration.

24.3.2.3 Digging Tool Geometry File

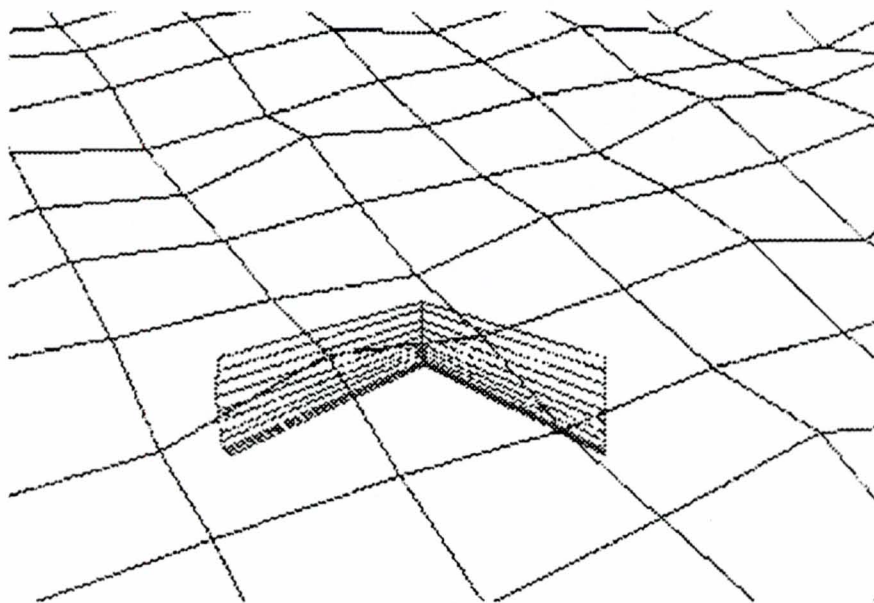


Figure 131. Wireframe model of breaching blade tool

The geometry file specifies the 3D model of the digging tool. A simple ASCII file format is used to describe the digging tool. The first line of the file contains the following data:

- *Mesh type*: 3 indicates a triangular mesh; 4 a rectangular mesh
- *Point count*: total number of points in the wireframe
- *Mesh size count*: total number of triangular or rectangular meshes
- *Normal_exist*: 1 indicates presence of plane normal information; 0 indicates none.

The remaining lines specify X, Y, Z location and normal information for the wireframe points. A sample file describing the breaching blade tool in Figure follows:

Add picture of wireframe model & file that describes it.

```
4 8 2 1
0.000000 0.000000 0.000000 norm -0.61709 -0.27650 0.736614
5.000000 2.000000 0.000000 norm 0.000000 0.000000 0.000000
5.000000 2.000000 2.000000 norm 0.000000 0.000000 0.000000
0.000000 0.000000 2.000000 norm 0.000000 0.000000 0.000000
5.000000 2.000000 0.000000 norm 0.000000 0.000000 0.000000
10.00000 0.000000 0.000000 norm 0.000000 0.000000 0.000000
10.00000 0.000000 2.000000 norm 0.000000 0.000000 0.000000
5.000000 2.000000 2.000000 norm 0.000000 0.000000 0.000000
```

The first line specifies that the digging tool model uses a rectangular mesh. It contains 8 points in the wireframe and consists of two rectangle meshes. Normal information is included for every point in the data file.

A modeling tool, such as MultiGen, can be used to generate the digging tool geometric model. With a few conversion utilities, the format described above can be achieved. Conversion steps follow:

Generate a geometric model of the digging tool in MultiGen.

Convert the Multigen FLT (V11.0 or 12.0) file into a WorldToolkit NFF V2.0 Sense8 file using *flt2nff*.¹

Convert the Sense8 format to the internal Digging Tool format using *nff2dig*.²

The Digging Tool geometry file can be easily created by hand for a simple tool shape.

24.3.3 Software Design

The digging tool model is implemented in the DiggingToolModel.

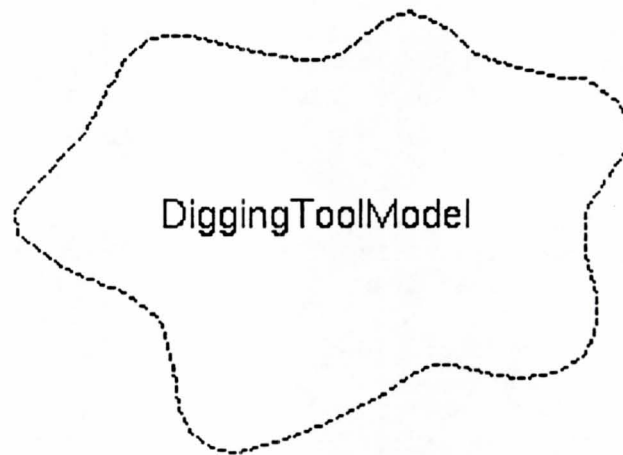


Figure 132. Booch diagram of digging tool model

24.3.4 Programmer's Guide

24.3.4.1 Class Definition

The implementer should become familiar with the DiggingToolModel specification to gain an understanding of the public methods provided by the class. The class structure is divided into the tool's geometric model, soil physical model features, and public digging methods. Specification of its public methods follows:

```
class DiggingToolModel
{
    public:
        DiggingToolModel(char *fname) ;// fname: The configuration file
        //name for the digging tool
        void initDigging(float step, float pstep, float
x_grid_step, float
        y_grid_step, int ns, int nt );// step: digging step for con-
stant digging.
```

1. This conversion utility was obtained from Digging Tool DTR and other DT project.

2. This conversion utility was developed for the Digging Tool DTR on the DT Project.

```

    // pstep: piling step for const digging.
    // x_grid_step: resolution of the terrain surface in x direc-
tion
    // y_grid_step: resolution of the terrain surface in y direc-
tion
    // ns: number of grid points in x direction
    // nt: number of grid points in y direction
    void initDigging(float step, float pstep,
        float xg_step, float yg_step, int ns, int nt,
        DynamicPara *dynamic, GeometryPara *geometry) ;
    // dynamic: The dynamic parameters needed for digging initial-
ization.
    // geometry: The geometric parameters needed for digging ini-
tialization.
    void digging(float new_depth, float old_height[][3], float
new_height[]) ;
    void place_digger(float depth, Point *dir,
Point *loc, Point *orient);
    void placeDigger( Point *loc);
    void placeDigger();
    void turnDigger(float rz) ;
    void moveDigger(float dist, float ddv, float evalution) ;
    void moveDigger(float newx, float newy) ;
    void drawDiggingTool( ) ;
    // new_depth: current digging depth
    // old_height: the queried patch from DTDB
    // new_height: modified patch in form of one-dimension data.
    // refer to the DTDB specification for more details about the
    // coners format for the patch
    void piling(float new_depth,float new_height[],float
swell_factor);
    // new_depth: current digging depth
    // new_height: modified path after piling in form of one-dimen-
sion
    // data.

    void newDepth(float depth);
    // depth: set the new digging depth
    void newOrient(float rx, float ry, float rz );
    // rx, ry, rz: The new orient angle
    void newVelocity(float v);
    // v: new velocity for the constant digging
    void newAccerlation(float a );
    // a: new accerlation for non-constant digging
    void newFcut(float f);
    // f: cutting force for the dynamic model
    void queryDiggerInfo(DIGGER_INFO *d_info) ;

} ;

```


24.3.4.2 Public Methods

24.3.4.2.1 DiggingToolModel(char *fname)

The constructor is called with the digging tool configuration file name. The constructor will read the configuration file and fill the class structure with data items in the file. For more description about the configuration file format, please refer to the file format specification.

24.3.4.2.2 void queryDiggerInfo(DIGGER_INFO *d_info)

This provides Digging Tool DTR clients access to private data in the *DiggingToolModel* such as the resistant force of the plowed soil. Another example is that a client may need to know patch coordinate to query and update DTDB and a demo may need to know the digging direction vector (ddv) to control the digging interactively.

```
void queryDiggerInfo(DIGGER_INFO *d_info);
typedef struct {
    float q_current_depth ;// The current digging depth
    float q_velocity ;      // The initial constant speed of digging
    float q_accerlation ;// The acceleration of digger
    float q_fcut ;// The cutting force of the digging tool
    float q_fresist; // The resist force of soil
    float q_mass_vel ;// The mass of the vehicle
    float q_ddv ;// The digging direction vector
    Point q_dlp1; // Digline point 1 (for old version)
    Point q_dlp2;
    Point init_pos ;
    float q_dstep ;// Current digging step size
    float q_pstep ;// Current piling step size
    float q_ul[2] ;// upleft of digging terrain patch
    float q_ur[2] ;// upright of digging terrain patch
    float q_lr[2] ;// lowright
    float q_ll[2] ;// lowleft
} DIGGER_INFO ;
```

24.3.4.2.3 void placeDigger(Point3 *loc), void placeDigger()

Place the digging tool into the terrain. This method can be treated as utility function and is optional. The application to use it to develop DTR client more easily. Of course, application may have its own way of placing digger, turning digger and etc.

24.3.4.2.4 void drawDiggingTool()

Rendering the digging tool. Since we have defined the digging tool wireframe format and supply some utilities to convert the modelling data, it's a pain for novel user to write drawing function for digging tool. We only supply the GL version of rendering for simplicity. Application can use self-defined rendering function instead calling this method.

It's the responsibility of the application to give the correct viewing, shading and other graphics operation.

**24.3.4.2.5 void initDigging(float step, float pstep, float x_grid_step, float y_grid_step, int ns, int nt),
void initDigging(float step, float pstep, float xg_step, float yg_step, int ns, int nt, DynamicPara *dynamic, GeometryPara *geometry) ;**

The digging and piling points are pre-calculated before digging and piling since in most case (An index of the points is stored in a lookup table for faster digging and piling operations. Anytime there is a change to one of the following parameters, initDigging() should be invoked:

- dstep: digging step size which is depend on the speed of digging tool of the attached vehicle or the presetting.
- pstep: piling step size which is depend on the soil type, the resistance force and other parameter.
- x_grid_step: grid size in x direction and is determined by the resolution of the terrain patch.
- y_grid_step: grid size in y direction and is determined by the resolution of the terrain patch.
- ns, nt: number of grid points in x and y direction respectively.
- dynamic: If the digging tool is used with other vehicle simulation, the dynamic parameters should be supplied to the digging tool model to calculate the resistant force, digging and piling distance.
- geometry: If the digging tool is used with other vehicle simulation, the geometry data structure will supply the required blade geometric info for digging, such as width of the blade, and depth for digging.

24.3.4.2.6 void digging(float new_depth, float old_height[][3], float new_height[])

Do the local digging on the terrain patch. The old terrain patch old_height is passed to the method and updated terrain patch is stored in new_height. The digging is based on the pre-calculated the index stored in digging data structure in DiggingToolModel and elevation of the affected point is replaced by the new_depth.

24.3.4.2.7 void piling(float new_depth, float new_height[], float swell_factor)

The piling() function will pile the soil in local terrain patch both in front and along the sides of the digging tool based on volume conservation. The affected patch new_height and the soil property, swell_factor are passed to the method and DiggingToolModel will determine the height of piling based on the volume reservation principle

24.3.4.2.8 void turnDigger(float rz)

This method controls the turning of the digging tool and is used to change the current digging direction. (This method can be treated as utility function is optional for application to use it to

develope DTR client more easily. Of course, application may have its own way of placing digger, turning digger and etc.).

24.3.4.2.9 void moveDigger(float dist, float ddv, float evalution) void moveDigger(float newx, float newy)

(This method can be treated as utility function is optional for application to use it to develope DTR client more easily. Of course, application may have its own way of placing digger, turning digger and etc.)

This method is used for application to move the digging tool while digging interactively. Specification of the input parameters: dist:how far the digging tool will be moved. ddv:the digging direction vector and is interpreted as angle. evalution: the relative evaluation compare with last move due to the different digging depth in each digging.

24.3.4.2.10 void newVelocity(float v) void newAccerlation(float a) void newFcut(float f)

This group of methods are used to set the physical properties of the digging tool during the digging process. Those parameters will be used to calculate the resistant force for the digging and piling. The default value or the initial value is set in configuration file.

newVelocity(v): Setting the new velocity of the digging tool or the velocity of the vehicle to where the digging tool is attached.

newAccerlation(a): Setting the new acceleration of the digging tool or the acceleration of the vehicle to where the digging tool is attached.

newFcut(f): Setting the cut force of the digging tool. Both the v, a and f will be used to compute the resistant force.

Application may not use those function and just set those parameters once in the configuration file. .

25.0 Coordinate Conversion

25.1 Introduction to Coordinate Conversion

This section introduces and reviews the algorithms that are used by the Entity Service for performing coordinate conversions. The Entity Service does not perform these coordinate conversions itself. Instead, all conversions are performed by an interface protocol called the Entity Service Client Interface. The client interface is used because each of the client programs that communicates with the Entity Service has its own internal format for storing entity and event state information. Any of these client formats may be quite different from the DIS-based format that the Entity Service uses. In order to provide clients with data in their own internal format, the client interface is designed to translate all state information into the client's format during every request for updates from the Entity Service. In addition, all of the state information generated by a client is translated into DIS format before being sent into the Entity Service.

25.2 Types of Coordinate Conversions

Three different generalized classes of state conversions are used to transform state information between the DIS state format and any of the varying formats provided for use by the clients. Depending on which state values need to be transformed, a different set of operations are performed. With the notable exception of transforming the orientation, all transformations are only dependant on one of the general classes of transformations. The transform methods are all reversible to insure two-way conversions.

25.2.1 Global Vector Conversions

Global vectors are used to describe quantities that are based only on the global or world coordinate system. These vectors deal only with points, i.e. the origin of the entity or event, without a concept of body axes or model extents. Position vectors prescribe the location of entity and event origins in global coordinates. Global velocity and acceleration vectors prescribe how an origin translates with respect to the global coordinate system. The exact structure of these vectors depends on which coordinate system is being used to model the geometry of the simulated environment.

One of the transformation classes used in the client interface handles differences in global, or world, coordinate systems. This class transforms global vectors between any of the various formats used for modeling databases or for standardizing networks. A variety of approaches are used in modeling databases for simulators. Many of these approaches vary quite a bit regarding what data structures they use and as to what the data structures represent. In addition, there is no guarantee that any databases will be modeled the same even when the exact same type of coordinate system is used. Therefore, this class is used to guarantee that all clients get the type of global data needed.

25.2.2 Body Vector Conversions

Body vectors are used to describe quantities that are dependent only on coordinate systems local to a body. These vectors deal with any values that are invariant with respect to the orientation of an entity or event. In other words, the value of a body vector does not change as the body it is attached to rotates; body vectors are only measured within a body coordinate system. Vectors such as translational velocity and acceleration can be resolved into body axes. Angular velocity and acceleration are almost always specified in terms of body axes. In addition, body vectors can be used for prescribing the motion of articulated parts or special effects. The exact structure of these vectors depends on exactly how the body coordinate system is set up for modeling the entity or event.

One of the transformation classes used in the client interface handles differences in body, or local, coordinate systems. This class transforms body vectors between any of the alternative approaches used for creating entity and event models. Most models are created with a very common, basic structure, that of a standard cartesian set of axes. There are two major ways these types of axes vary between models. First, the axes can be oriented quite differently within the model. Second, the origin used for the body axes can be set up almost anywhere, including outside of the model. In addition, there are other methods for modeling that do not use cartesian coordinates at all. This class is used to guarantee that all clients get the type of model data needed.

25.2.3 Orientation Conversions

Orientation is a special case. Since orientation is not a vector quantity, it cannot be stored as a vector or manipulated with vector methods. A variety of approaches have been developed for storing orientation. However, most of these approaches produce behavior that is not well-defined or predictable. Therefore, additional steps are required in order to convert orienting between state formats.

The behavior of orientation is made manageable by storing it as a rotation matrix. On each side of the client interface, special classes are used to transform orientation into and out of rotation matrices. Orientation is inputted to the client interface as a set of values and a format. This is read in by a matrix encoder that produces a rotation matrix with identical orientation to the input values. Next, this new rotation matrix is processed by the two major conversion classes, managed once like a global vector and once like a body vector. This is possible because the rows of a rotation matrix act in the same way as the elements of a vector. Finally, the resultant rotation matrix is read into a matrix decoder that produces equivalent orientation values in another format. This other format is the one needed on the other side of the interface.

25.2.4 Layout of the Client Interface

The classes that are discussed in the prior sections form the basis of the coordinate conversions used by the Entity Service Client Interface. The next figure is a schematic diagram showing the layout of all of the conversion elements and how they each process state data. Note that the layout may need to be redesigned as more coordinate systems are added.

25.3 Types of Coordinate Systems

The types state formats used in the Entity Service Client Interface fall into one of three general categories. The first category deals with the coordinate systems used for the modeling of databases. The second one deals with coordinate systems used to model any entities and events. The third type deals with orientation, which is distinct from the other two because orientation is the quantity that describes the relationship between the global and body coordinate systems. In the following sections, all of these general categories are discussed and examples of formats in common usage are introduced. It should be noted that this listing is not complete and other alternatives do exist. However, these formats are the ones that will most likely be incorporated into the Client Interface.

25.3.1 Types of Database Coordinate Systems

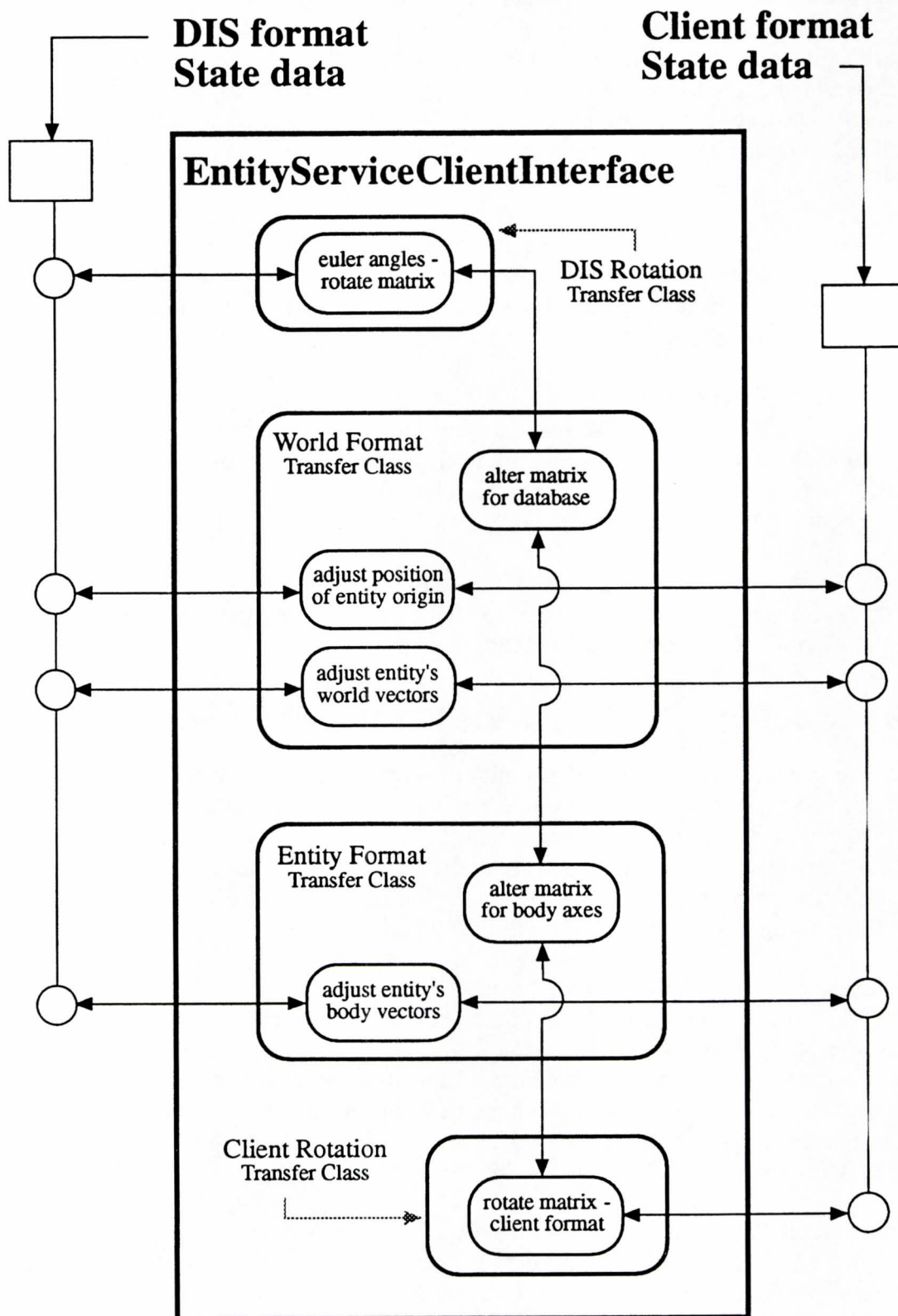
The coordinate system that is used to model a database acts as a reference from which all points in the database, moving or static, can be measured. Selection of a database coordinate system thereby defines the basis of all data structures that can be used to model the database and all entities and events on it.

The DIS geocentric coordinate system is a cartesian (X, Y, Z) coordinate system with its origin placed at the center of the earth. The X-axis of this system is directed from the origin out through the Prime Meridian. The Y-axis then points out ninety degrees east of the X-axis, or through longitude ninety degrees east. The resulting X-Y plane becomes the equatorial plane. Finally, the geocentric Z-axis goes through the true north pole. This system was chosen for DIS in order to allow for modeling of the entire planet without the complexities of a non-cartesian system. The surface of the earth is modeled according to the WGS84 spheroid, but this is not a rigid constraint. Geocentric data structures do not require a particular surface representation.

Not many databases are modeled with a geocentric coordinate system. Therefore, most simulators must convert into and out of geocentric coordinates when interfacing with a DIS network. A major advantage of using the Entity Service Client Interface is that it can provide clients with state information in the same database coordinate system as they use internally. There are several types of coordinate systems in common usage.

The geodetic coordinate system is a common designation that is used for modeling the entire surface of the earth. Geodetic coordinates utilize the familiar values of latitude, longitude, and altitude to denote position. Velocity and acceleration are specified by using components going east-west, north-south, and up-down.

Most other database conventions utilize a flat database with a cartesian coordinate system placed on the surface. To account for the curvature of the surface of the planet, these flat databases can be mathematically wrapped around the planet. Wrapping of a flat database allows the flat coordinates to be transformed into a global coordinate system. A simulator can thus use a flat database and relate its values to other databases that are also placed on the planet. These coordinate systems are much more convenient when large sections of the planet's surface are not required in a simulation.



0703-6963

A convention which accounts for the entire surface of the earth while using the flat earth approach is the UTM (upper transverse mercator) map projection system. It uses a set of standardized, flat earth maps that wrap around certain parts of the earth. All of these maps have their origin located at the south-west corner, with the positive x-axis going east and the positive y-axis heading north. The placement and wrapping of each of these maps has been developed and regulated by the military. Each map is coded with a convention that allows its placement on the surface to be quickly determined.

A logical expansion of the UTM approach is using custom defined, flat earth maps that can be arbitrarily placed onto the surface of the earth at any location. A map such as this could also be oriented in a different manner from the UTM standard. In addition, the origin could be located in a different part of the database.

A flat earth database does not necessarily have to be wrapped around the surface of the planet. By only modeling a relatively small surface area, the need to wrap the database around the planet can be reduced. If the longest extent of the database is only a few miles, the curvature of the earth will be minor and the need for wrapping will be negligible. This simplification allows all conversions to take place between two statically related cartesian coordinate systems.

There are, of course, other database coordinate systems in use. However, the types that are discussed in this section are the ones most likely to be incorporated into the Entity Service Client Interface. Others may follow if they are needed.

25.3.2 Types of Entity/Event Coordinate Systems

The coordinate system utilized to model an entity or event determines how they can be modeled. Global quantities are given only in terms of the origin of an entity or event. The model coordinate system is attached to this origin and is used to define all of the points in the body. It also provides a reference for all of the vectors attached to the body. Selecting a standard set of body axes for all entities in a simulation simplifies the simulation, but it is not essential. However, almost all simulators do utilize a single body axes convention for all of their models.

Most simulators utilize a variation of a simple prototype set of axes to model all of their entities and events. The prototype consists of a set of six axes, or three pairs of twin directional axes. These pairs represent forward-to-reverse, left-to-right, and top-to-bottom directions relative to the body. One direction out of each pair is selected. Then the axes are labeled as (x,y,z), (1,2,3), or something similar. Body axes are not always set up in this manner, but it is the most common approach by far. In fact, cartesian axes do not have to be used at all; even non-orthogonal axes can be used. Most simulators, however, use this basic prototype.

Another modeling difference concerns how the origin is placed within the body. A standard convention for ground vehicles places the origin at the base of the vehicle, at the middle of the vehicle's length and width. Almost all ground vehicle models use this as an origin. However, as more types of entities are modeled more often, there will be a need to correct for offset origins, especially in aircraft and seacraft.

25.3.3 Types of Orientation Formats

Orientation is used to define the relationship between a database coordinate system and a model coordinate system. As such, it cannot be a vector quantity. Instead, what an orientation represents is a transformation between coordinate systems. This complicated nature has led to a variety of approaches for storing and transmitting orientation. Many of these are in common usage and need to be accounted for.

Euler angles are the most common representation of orientation. They represent a sequence of three rotations around the body axes of an entity. By performing three distinct rotations, any possible orientation may be achieved. However, these euler angles are not easy to manipulate for many reasons. One, euler sequences are not vectors, even though many people misinterpret them as such. In addition, there are at least twelve distinct sets of angles in common usage, even though many people believe there is only one. To avoid confusion, it is important that euler angles be specified as clearly as possible.

Another major format is euler parameters, or quaternions. While not a vector, the quaternion is a well behaved data structure with well-defined rules for manipulating them. Quaternions have four elements that are constructed using the magnitude of rotation and the normal vector that the rotation occurs. These quantities can be manipulated in a reasonable manner and do not suffer from singularities and discontinuities as euler angles do. In addition, there is only one quaternion for any given set of axes, reducing confusion and helping standardization.

Rotation matrices are the simplest structure for managing orientation. Since most of the alternate structures eventually have to be turned into a rotation matrix, using one of them directly saves computations. However, the size of the structure increases because a rotation matrix has nine elements. This is offset somewhat by the extremely well behaved nature of rotation matrices.

Many other orientation formats exist. However, including them all into the design of the client interface is a large task. Other formats can be added if they are needed. The structure of the interface is abstract enough to allow almost any existing orientation format to be used as input or output.

25.3.4 Coordinate Systems used in the Client Interface

Due to time restrictions, all of the coordinate systems in use cannot be included at one time. For development purposes a set of basic structure were selected for inclusion in the initial client interface. This basic set of formats satisfies all of the requirements of the Dynamic Terrain Project. Other formats will probably be added after the entire system has been fully tested.

The only database format that is initially supported by the client interface will be the small scale, flat earth model. The model for entities and events assumes body axes derived from the basic set of common, orthogonal axes. All of these will be supported but the body origins will be constrained to be constant. Initially, orientation will be limited to rotation matrices and some of

the possible euler angles.

25.4 Conversions between Coordinate Systems

Three major sets of conversions take place in the Entity Service Client Interface. The first set is comprised of converting position and global vectors. The second set is dedicated to conversion of body vectors. Finally, the third set accounts for all of the things that affect orientation. Each of these sets of conversions is introduced and discussed in the following sections.

25.4.1 Position and Global Vector Conversions

Two database formats will be accounted for by the initial client interface. The DIS geocentric system and the simplest other case, a flat earth model with a small surface area. These two types of models are actually just alternate sets of cartesian axes that are offset and rotated from each other. Since these two sets of axes have a fixed relationship, much of the conversion work only has to be performed once and can then be continually reused. The following figure demonstrates the relationship between the DIS geocentric axes and the small scale surface axes. The mathematical relationships between the axes are shown on the next page, along with the conversion methods required to go between the two sets of coordinate systems.

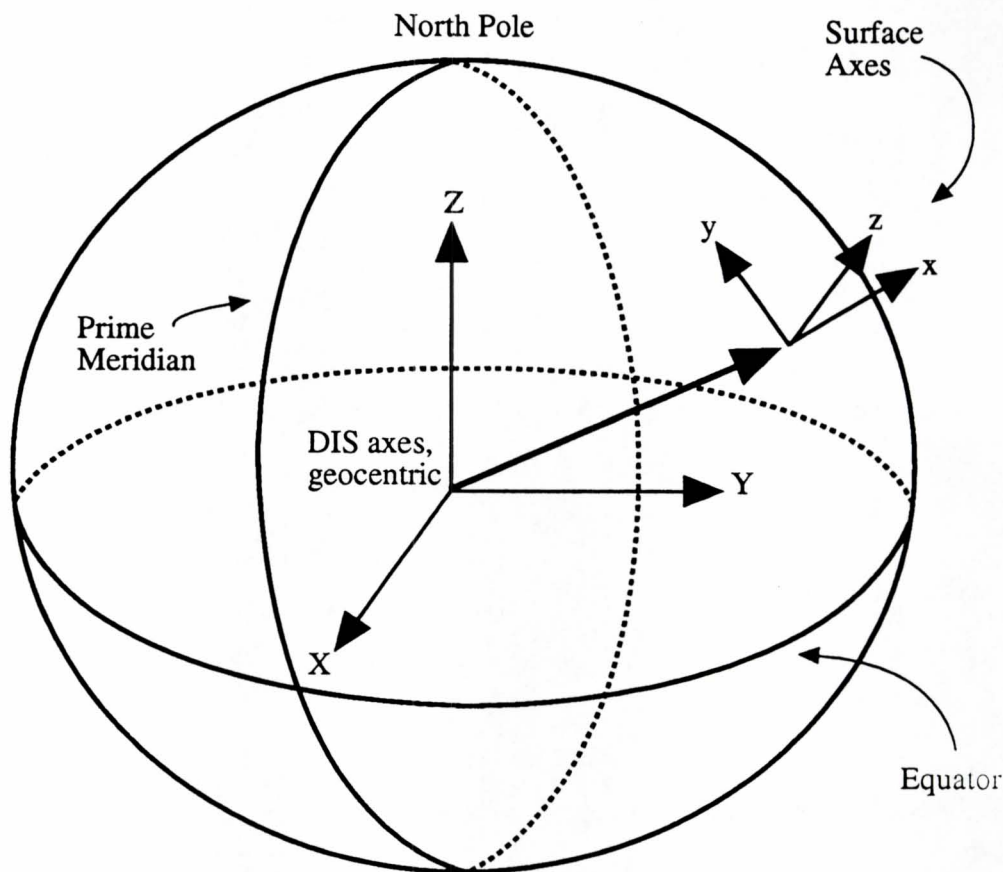
The offset vector between the DIS origin and the origin of the surface axes is given in terms of the geocentric coordinates of the origin of the surface axes.

$$\begin{bmatrix} \text{offset X} \\ \text{offset Y} \\ \text{offset Z} \end{bmatrix}_{DIS} = \begin{bmatrix} \text{DIS X Coordinate of Surface Origin} \\ \text{DIS Y Coordinate of Surface Origin} \\ \text{DIS Z Coordinate of Surface Origin} \end{bmatrix} \quad (104)$$

The rotation matrix between the DIS axes and the surface axes is given in terms of the geocentric rotation simple matrix of the surface axes.

$$\begin{bmatrix} t_{XX} & t_{XY} & t_{XZ} \\ t_{YX} & t_{YY} & t_{YZ} \\ t_{ZX} & t_{ZY} & t_{ZZ} \end{bmatrix}_{DIS} = [\text{DIS Rotation Matrix of Surface Axes}] \quad (105)$$

Given these two sets of constants, it is straightforward to convert values between the DIS geocentric system and the surface axes. The conversions from the DIS geocentric system into surface coordinates are detailed below. The conversions from surface system into DIS are shown



0703-6964

in the following equations.

Global vectors such as the global velocity and acceleration are converted with a simple matrix multiplication.

$$\text{Global Surface Velocity} = \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_{\text{Surface}} = \begin{bmatrix} t_{XX} & t_{XY} & t_{XZ} \\ t_{YX} & t_{YY} & t_{YZ} \\ t_{ZX} & t_{ZY} & t_{ZZ} \end{bmatrix}_{\text{DIS}} \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_{\text{DIS}} \quad (106)$$

$$\text{Global Surface Acceleration} = \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix}_{\text{Surface}} = \begin{bmatrix} t_{XX} & t_{XY} & t_{XZ} \\ t_{YX} & t_{YY} & t_{YZ} \\ t_{ZX} & t_{ZY} & t_{ZZ} \end{bmatrix}_{\text{DIS}} \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix}_{\text{DIS}} \quad (107)$$

Position conversions require a two step process.

$$\text{Delta Vector} = \begin{bmatrix} \Delta X \\ \Delta Y \\ \Delta Z \end{bmatrix}_{DIS} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{DIS} - \begin{bmatrix} \text{offset X} \\ \text{offset Y} \\ \text{offset Z} \end{bmatrix}_{DIS} \quad (108)$$

$$\text{Surface Position} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{Surface} = \begin{bmatrix} t_{XX} & t_{XY} & t_{XZ} \\ t_{YX} & t_{YY} & t_{YZ} \\ t_{ZX} & t_{ZY} & t_{ZZ} \end{bmatrix}_{DIS} \begin{bmatrix} \Delta X \\ \Delta Y \\ \Delta Z \end{bmatrix} \quad (109)$$

Reverse conversions, transforming from surface coordinates into DIS geocentric coordinates, are shown below.

$$\text{Global DIS Velocity} = \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_{DIS} = \begin{bmatrix} t_{XX} & t_{XY} & t_{XZ} \\ t_{YX} & t_{YY} & t_{YZ} \\ t_{ZX} & t_{ZY} & t_{ZZ} \end{bmatrix}_{DIS}^T \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_{Surface} \quad (110)$$

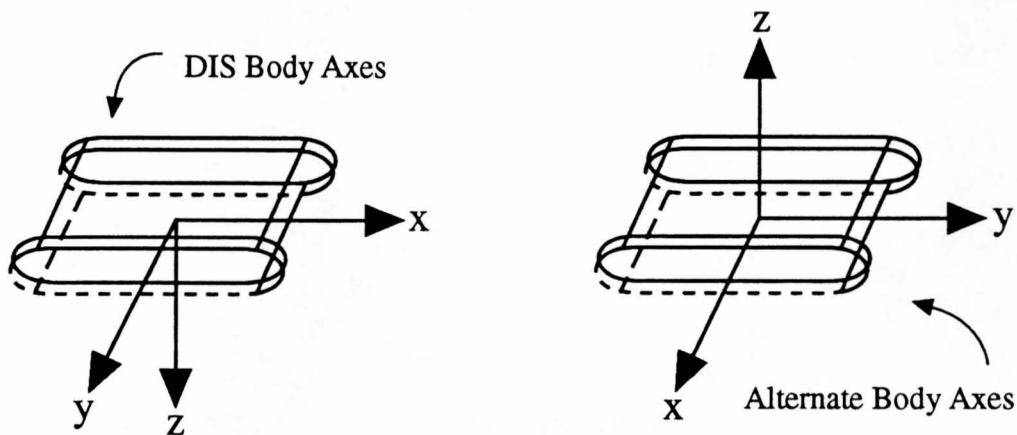
$$\text{Global DIS Acceleration} = \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix}_{DIS} = \begin{bmatrix} t_{XX} & t_{XY} & t_{XZ} \\ t_{YX} & t_{YY} & t_{YZ} \\ t_{ZX} & t_{ZY} & t_{ZZ} \end{bmatrix}_{DIS}^T \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix}_{Surface} \quad (111)$$

$$\text{Delta Vector} = \begin{bmatrix} \Delta X \\ \Delta Y \\ \Delta Z \end{bmatrix} = \begin{bmatrix} t_{XX} & t_{XY} & t_{XZ} \\ t_{YX} & t_{YY} & t_{YZ} \\ t_{ZX} & t_{ZY} & t_{ZZ} \end{bmatrix}_{DIS}^T \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{Surface} \quad (112)$$

$$\text{DIS Position} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{DIS} = \begin{bmatrix} \Delta X \\ \Delta Y \\ \Delta Z \end{bmatrix} + \begin{bmatrix} \text{offset X} \\ \text{offset Y} \\ \text{offset Z} \end{bmatrix}_{DIS} \quad (113)$$

25.4.2 Body Vector Conversions

Conversions between different entity/event coordinate systems is restricted to only the basic system of modeling. This means conventional, orthogonal body axes having the same body origin. A common example of this type of difference is shown in the following figure.



0703-6965

The actual procedure utilized by this part of the conversion process is very simple, but still quite important. Since the exact conversion performed depends more on straight logic than math. Therefore, one example of converting body-axes vectors will be shown, using the case illustrated in the figure above.

$$\text{Alternate Angular Velocity} = \begin{bmatrix} \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix}_{Alt} = \begin{bmatrix} \omega_Y \\ \omega_X \\ -\omega_Z \end{bmatrix}_{DIS} \quad (114)$$

$$\text{Alternate Body Velocity} = \begin{bmatrix} v_X \\ v_Y \\ v_Z \end{bmatrix}_{Alt} = \begin{bmatrix} v_Y \\ v_X \\ -v_Z \end{bmatrix}_{DIS} \quad (115)$$

$$\text{Alternate Body Acceleration} = \begin{bmatrix} a_X \\ a_Y \\ a_Z \end{bmatrix}_{Alt} = \begin{bmatrix} a_Y \\ a_X \\ -a_Z \end{bmatrix}_{DIS} \quad (116)$$

The transformations are reversed by exactly the same process. The exact same steps can be used to transform body axes in either direction.

$$\text{Alternate Angular Velocity} = \begin{bmatrix} \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix}_{DIS} = \begin{bmatrix} \omega_Y \\ \omega_X \\ -\omega_Z \end{bmatrix}_{Alt} \quad (117)$$

$$\text{Alternate Body Velocity} = \begin{bmatrix} v_X \\ v_Y \\ v_Z \end{bmatrix}_{DIS} = \begin{bmatrix} v_Y \\ v_X \\ -v_Z \end{bmatrix}_{Alt} \quad (118)$$

$$\text{Alternate Body Acceleration} = \begin{bmatrix} a_X \\ a_Y \\ a_Z \end{bmatrix}_{DIS} = \begin{bmatrix} a_Y \\ a_X \\ -a_Z \end{bmatrix}_{Alt} \quad (119)$$

Note that the prior six equations are only examples. However, the process will always take place the same way. Rows of the vector may be swapped with each other or with the negation of each other. They may also be simply negated or just left alone. As is shown above, setting up this part of the conversion is quite simple. All that is required is storing whichever axes need swapping and/or negating. No complicated mathematical functions are required.

25.4.3 Conversion of Orientation

Orientation is the most difficult state quantity to transform between two coordinate systems due to the additional steps that are required. The first step that needs to be performed is calculating a rotation matrix from the native (input) orientation format. Then this rotation matrix is modified to account for any differences in both global axes and body axes. Finally, the resultant rotation matrix is converted into whatever format is desired for the output. These general step function for transforming orientation from DIS into client format and vice versa. The steps are described below, starting from the DIS side of the system.

First, a rotation matrix is created that is equivalent to the DIS euler angles.

$$\text{DIS Euler Angles} = (\psi, \theta, \phi) \quad (120)$$

$$\text{DIS Rotation Matrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{DIS} = \text{Function}(\psi, \theta, \phi) \quad (121)$$

Next, the matrix is modified to account for the rotation of the surface axes versus the geocentric axes.

$$\text{Temp Matrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{Temp} = \begin{bmatrix} t_{XX} & t_{XY} & t_{XZ} \\ t_{YX} & t_{YY} & t_{YZ} \\ t_{ZX} & t_{ZY} & t_{ZZ} \end{bmatrix}_{DIS} \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{DIS} \quad (122)$$

Then the matrix undergoes the swapping and negation needed to account for how the body axes differ. Note that the example setup from the prior section is utilized in for this equation. This is for example purposes only; the equations will be different in many actual cases.

$$\text{Client Rotation Matrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{\text{Client}} = \begin{bmatrix} r_{YX} & r_{YY} & r_{YZ} \\ r_{XX} & r_{XY} & r_{XZ} \\ -r_{ZX} & -r_{ZY} & -r_{ZZ} \end{bmatrix}_{\text{Temp}} \quad (123)$$

Finally, the orientation is converted into the format utilized by the client. Note that this is left open for many different types.

$$\text{Client Orientation Format} = \text{Function} \left(\begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{\text{Client}} \right) \quad (124)$$

These steps can be easily reversed to transform client orientation into DIS format orientation. The steps are shown below.

$$\text{Client Rotation Matrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{\text{Client}} = \text{Function}(\text{Client Orientation}) \quad (125)$$

$$\text{Temp Matrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{\text{Temp}} = \begin{bmatrix} r_{YX} & r_{YY} & r_{YZ} \\ r_{XX} & r_{XY} & r_{XZ} \\ -r_{ZX} & -r_{ZY} & -r_{ZZ} \end{bmatrix}_{\text{Client}} \quad (126)$$

$$\text{DIS Rotation Matrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{\text{DIS}} = \begin{bmatrix} t_{XX} & t_{XY} & t_{XZ} \\ t_{YX} & t_{YY} & t_{YZ} \\ t_{ZX} & t_{ZY} & t_{ZZ} \end{bmatrix}_{\text{DIS}}^T \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{\text{Temp}} \quad (127)$$

$$\text{DIS Euler Angles} = (\psi, \theta, \phi) = \text{Function} \left(\begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{\text{DIS}} \right) \quad (128)$$

Note that details were left out about how the euler angles and other client formats are transformed into a matrix and how the matrices were decomposed to produce a format that the recipient can understand. These were left out because of their complexity and also because they are beyond the scope of this document.

26.0 Dead Reckoning

26.1 Introduction

This paper briefly introduces and reviews the algorithms implemented to perform dead-reckoning in the Entity Service. These algorithms are implemented in order to reduce the number of network packets that need to be transmitted. By sending dead reckoning parameters in each update, the position and orientation of each entity can be extrapolated between packets. Thus, the Entity Service can use the dead reckoned models as a data pool for its clients. The clients only have to request state information on an entity from the Entity Service, they do not have to be aware of the network. The Entity Service maintains the state of all entities for its clients. The dead reckoning algorithms simply allows the Entity Service to provide these states in a continuous manner.

The dead reckoning algorithms are based on the required dead reckoning models established by DIS standard 2.0.X and other supporting documents. However, some of the models required specific pieces of information that are lacking from the standard and the other documents. In these cases, decisions were made based on standard approaches to kinematic modeling and simulation used in mechanical engineering. The general dead reckoning types required by DIS are discussed in the next section.

26.2 Dead Reckoning with DIS Standards

Compliance with versions 2.0 and 2.0.3 of the DIS standard requires simulators to support nine different types of dead reckoning for networked entities. The purpose of these methods is to reduce the network traffic by providing a means for separate, asynchronous systems to predict the motion of remote entities between update packets. The entity state PDUs of each entity contain a field that specifies which dead reckoning algorithm must be used by all other systems for dead reckoning that particular entity. In addition, there are fields containing all of the necessary kinematic information required by the specified dead reckoning algorithm.

The dead reckoning algorithms are specified with three major variables. The first of these denotes whether or not the entity's body axes are rotating. The second variable indicates what (equation) order of algorithm is used for dead reckoning. What axes the velocity and acceleration are specified in, are prescribed in the third variable.

Parameter One = F, fixed body axes (no entity rotation)

Parameter One = R, rotating body axes.

Parameter Two = P, "position" (constant velocity) dead reckoning

Parameter Two = V, "velocity" (constant acceleration) dead reckoning

Parameter Three = W, geocentric world axes for velocity and acceleration

Parameter Three = B, body axes velocity and acceleration

The variations of these three parameters have been grouped and enumerated for the DIS standard. All of the currently acceptable types of dead reckoning methods for DIS are listed below:

<u>Field Value</u>	<u>Dead Reckoning Model (DRM)</u>
0	Other (future) Algorithms
1	Static (no dead-reckoning)
2	DRM (F, P, W)
3	DRM (R, P, W)
4	DRM (R, V, W)
5	DRM (F, V, W)
6	DRM (F, P, B)
7	DRM (R, P, B)
8	DRM (R, V, B)
9	DRM (F, V, B)

The next sections contain mathematical breakdowns and implementations for nine of the DIS dead reckoning models. The exception is Algorithm Zero, which is the reserved, "other" algorithm. It is intended for special cases that may be needed by unusual applications or for future requirements that have not yet been foreseen.

For the nine primary algorithms, the steps performed by Entity Service are broken down into two parts. First are initial steps that are performed once, upon receipt of a new PDU for a particular entity. This establishes a valid reference state for that entity. Then the specified dead reckoning algorithm extrapolates from this state to a new predicted state every time step. This continues until a new PDU is received.

At that point, the new PDU is treated as the new valid state. This new state will most probably conflict with the state of the dead reckoned entity. This is because the state of the true entity is allowed to vary within certain tolerances from the dead reckoned model. If this were not done, dynamic entities would be sending out PDUs in almost every one of their simulation frames. This would negate the major benefit of dead reckoning, namely reducing network traffic. Therefore, the source of each entity is required to dead reckon itself. When the differences between the local ghost state and the true entity state exceeds the tolerance (independently selected for each exercise), another PDU will be sent out. This is why the state of the dead reckoned entities conflicts with new PDUs.

Handling the conflict between a new PDU and the current dead reckoned state has not yet been specified in the DIS standards. For developmental purposes, the Entity Service originally jumped, or "teleported," entities from their current state to the new state. This technique of snapping the entity state is simple and easy to implement, but it can cause problems for more advanced applications. In particular, certain DT applications that track entities, such as the Track DTR and the Minefield DTR, suffer from discontinuity problems when entities snap from one state to another. In order to avoid these problems, it was decided that the Entity Service would provide some type of smoothing of the state when new PDUs are received. Since there is currently no standard for smoothing, a couple of approaches were tried and implemented, in addition to snapping, which was also left in as an option. The smoothing algorithms are discussed following the next three sections, which discuss standard DIS dead reckoning.

26.3 Kinematic Data Fields in DIS

A DIS Entity State PDU contains five fields of kinematic data. When a new PDU is received, the values in these fields are read in from the PDU and copied into equivalent fields within the appropriate dead reckoning model. These five sets of data are described in the following four sections.

26.3.1 Position

In the DIS standard, all positions are measured from an origin placed at the center of the earth. A cartesian coordinate system placed at this origin creates the DIS geocentric axes. The DIS geocentric X-axis points to the Prime Meridian, the Y-axis points to ninety degrees eastward of the X-axis, and the Z-axis points through the North Pole. The units of measure are meters.

$$\text{Position} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (129)$$

26.3.2 Orientation

In the DIS standard, entity orientation is specified in terms of one of the sequences of euler angles. The DIS euler angle sequence rotates between the DIS geocentric axes and the DIS standard body axes. The DIS geocentric axes are introduced and discussed in the prior section. The DIS standard for entity body axes is as follows: the entity's x-axis points out the front of the vehicle, the y-axis out the "starboard" side, and the z-axis down through the bottom of the vehicle. This set of body axes is commonly used by engineers in the aerospace industry. This set of body axes is utilized for all body expressions of the entity's state information.

The actual sequence of euler angles chosen by DIS is also most commonly used in the aerospace industry. The sequence goes as follows: the first of the euler angle rotations, psi or Ψ , occurs around the vehicle's z-axis, then the second rotation, theta or θ , is about the newly rotated y-axis, and finally, the third rotation, phi or ϕ , around the doubly rotated x-axis. All of these rotations occur counter-clockwise.

$$\text{Orientation} = [\Psi \ \theta \ \phi] \quad (130)$$

26.3.3 Angular Velocity

In DIS, entities can specify their angular velocity using a body axes vector. This vector passes through the entity's origin and is resolved in terms of the entity's body axes. The units used for angular velocity are radians per second.

$$\text{Angular Velocity} = \begin{bmatrix} \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix} \quad (131)$$

26.3.4 Translational Velocity and Acceleration

In DIS, entities can specify translational velocity and acceleration by using either world axes or body axes vectors, depending on dead reckoning parameter three. These vectors go through the entity's origin. The units for velocity are meters per second. The units for acceleration are meters per second squared.

$$\text{Translational Velocity} = \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix} \quad (132)$$

$$\text{Translational Acceleration} = \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix} \quad (133)$$

26.4 Implementation of Dead Reckoning Models

The full implementation of each dead reckoning model is introduced and described in the following ten sections, with each of the models listed and discussed separately. The algorithms are shown in their entirety, with the except of three common methods that are used to manage orientation. Their description is found further in the document.

26.4.1 Algorithm Zero, "Other" Model

This algorithm is reserved for use by "other" dead reckoning models that may be required for future systems of networking requirements.

26.4.2 Algorithm One, Static Model

In this model, the kinematic data fields are read in from the PDU and kept constant until the next PDU is received. This algorithm represents an unmoving or a parked entity. The entity's state values remain constant until another PDU is received.

26.4.3 Algorithm Two, Constant Global Velocity with

Fixed Axes

Algorithm two models a non-rotating entity that is translating at constant velocity, specified in world coordinates. Over each time step, simple first order dead reckoning of entity position is the only operation performed. The orientation and translational velocity remain constant. Angular velocity and translational acceleration are zero.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_t + \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix} \Delta t \quad (134)$$

26.4.4 Algorithm Three, Constant Global Velocity with Rotating Axes

Algorithm three models almost the same case as algorithm two. The difference is that the entity is also allowed to rotate. Position is dead reckoned as above, because the world axes velocity is decoupled from the orientation. Translational and angular velocity remain constant and translational acceleration is zero.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_t + \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix} \Delta t \quad (135)$$

$$\begin{bmatrix} \Psi \\ \theta \\ \phi \end{bmatrix} = \text{Dead Reckon} \left(\begin{bmatrix} \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix} \right) \quad (136)$$

26.4.5 Algorithm Four, Constant Global Acceleration with Rotating Axes

Algorithm four models almost the same case as algorithm three. The differences between the two are that a second order equation is used to dead reckon the position and that the translational velocity must also be dead reckoned, using a first order equation. It is very important to dead reckon the position before dead reckoning the velocity. The orientation is dead reckoned the same way as it is in the prior method. The translational acceleration is now a constant the same as the angular velocity.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_t + \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_t \Delta t + \frac{1}{2} \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix} \Delta t^2 \quad (137)$$

$$\begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_t + \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix} \Delta t \quad (138)$$

$$\begin{bmatrix} \Psi \\ \theta \\ \phi \end{bmatrix} = \text{Dead Reckon} \left(\begin{bmatrix} \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix} \right) \quad (139)$$

26.4.6 Algorithm Five, Constant Global Acceleration with Fixed Axes

Algorithm five is actually a simplified version of algorithm four. The difference is that orientation does not have to be modeled like it does in algorithm five. Second order equations are still used to dead reckon the position with first order ones for translational velocity. The translational acceleration is still constant. However, the orientation is now a constant again and the angular velocity is zero.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_t + \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_t \Delta t + \frac{1}{2} \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix} \Delta t^2 \quad (140)$$

$$\begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_t + \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix} \Delta t \quad (141)$$

26.4.7 Algorithms Six, Seven, Eight, and Nine

These four algorithms specify their translational velocity and acceleration in terms of body axes. While body axes are more convenient for many reasons, in order to perform dead reckoning, the velocity and acceleration must be converted into world axes terms. This is done using a rotation matrix that represents the rotation of the body axes relative to the world axes. This matrix is cre-

ated with the orientation methods described further in the document. For convenience, the matrix is demonstrated below.

$$\begin{bmatrix} Q \\ Q \\ Q \end{bmatrix}_{Body} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} \begin{bmatrix} Q \\ Q \\ Q \end{bmatrix}_{World} \quad (142)$$

$$\begin{bmatrix} Q \\ Q \\ Q \end{bmatrix}_{World} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}^T \begin{bmatrix} Q \\ Q \\ Q \end{bmatrix}_{Body} \quad (143)$$

26.4.8 Algorithm Six, Constant Body Velocity with Fixed Axes

Algorithm six is almost the same as algorithm two. The only difference is that the velocity needs to be transformed from body axes components into world components. This transformation is performed using the current rotation matrix. The position can now be dead reckoned with the same first order dead reckoning model. The orientation and the translational velocity remain constant. The angular velocity and translational acceleration are zero.

$$\begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} = \text{Generate Matrix} \left(\begin{bmatrix} \Psi & \theta & \phi \end{bmatrix} \right) \quad (144)$$

$$\begin{bmatrix} \text{World } V_X \\ \text{World } V_Y \\ \text{World } V_Z \end{bmatrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix} \quad (145)$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_t + \begin{bmatrix} \text{World } V_X \\ \text{World } V_Y \\ \text{World } V_Z \end{bmatrix} \Delta t \quad (146)$$

26.4.9 Algorithm Seven, Constant Body Velocity with

Rotating Axes

Algorithm seven is similar to algorithm six, except for the fact that the body is now allowed to rotate. This rotation, combined with the body velocities, generates a body axes acceleration, despite the fact that no explicit acceleration exists. Both the body axes terms are converted into world axes terms for second order dead reckoning of position. Then the orientation of the body is dead reckoned. The translational and angular velocities remain constant; translational acceleration is also constant.

$$\begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} = \text{Generate Matrix} \left(\begin{bmatrix} \Psi & \theta & \phi \end{bmatrix} \right) \quad (147)$$

$$\begin{bmatrix} \text{effective } a_X \\ \text{effective } a_Y \\ \text{effective } a_Z \end{bmatrix} = \begin{bmatrix} V_Z \omega_Y - V_Y \omega_Z \\ V_X \omega_Z - V_Z \omega_X \\ V_Y \omega_X - V_X \omega_Y \end{bmatrix} \quad (148)$$

$$\begin{bmatrix} \text{World } A_X \\ \text{World } A_Y \\ \text{World } A_Z \end{bmatrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} \begin{bmatrix} \text{effective } a_X \\ \text{effective } a_Y \\ \text{effective } a_Z \end{bmatrix} \quad (149)$$

$$\begin{bmatrix} \text{World } V_X \\ \text{World } V_Y \\ \text{World } V_Z \end{bmatrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix} \quad (150)$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_t + \begin{bmatrix} \text{World } V_X \\ \text{World } V_Y \\ \text{World } V_Z \end{bmatrix}_t \Delta t + \frac{1}{2} \begin{bmatrix} \text{World } A_X \\ \text{World } A_Y \\ \text{World } A_Z \end{bmatrix}_t \Delta t^2 \quad (151)$$

$$\begin{bmatrix} \Psi & \theta & \phi \end{bmatrix} = \text{Dead Reckon} \left(\begin{bmatrix} \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix} \right) \quad (152)$$

26.4.10 Algorithm Eight, Constant Body Acceleration

with Rotating Axes

Algorithm eight models almost the same case as algorithm seven. The differences between the two are that an explicit body axes acceleration exists. This alters the effective body axes acceleration. It also requires another step be inserted for dead reckoning of the body axes velocity. Otherwise, it is the exact same as algorithm seven. The position, the translational velocity, and the orientation are all dead reckoned. Translational acceleration and angular velocity are constants.

$$\begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} = \text{Generate Matrix} \left(\begin{bmatrix} \Psi & \theta & \phi \end{bmatrix} \right) \quad (153)$$

$$\begin{bmatrix} \text{effective } a_X \\ \text{effective } a_Y \\ \text{effective } a_Z \end{bmatrix} = \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix} + \begin{bmatrix} V_Z \omega_Y - V_Y \omega_Z \\ V_X \omega_Z - V_Z \omega_X \\ V_Y \omega_X - V_X \omega_Y \end{bmatrix} \quad (154)$$

$$\begin{bmatrix} \text{World } A_X \\ \text{World } A_Y \\ \text{World } A_Z \end{bmatrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} \begin{bmatrix} \text{effective } a_X \\ \text{effective } a_Y \\ \text{effective } a_Z \end{bmatrix} \quad (155)$$

$$\begin{bmatrix} \text{World } V_X \\ \text{World } V_Y \\ \text{World } V_Z \end{bmatrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix} \quad (156)$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_t + \begin{bmatrix} \text{World } V_X \\ \text{World } V_Y \\ \text{World } V_Z \end{bmatrix}_t \Delta t + \frac{1}{2} \begin{bmatrix} \text{World } A_X \\ \text{World } A_Y \\ \text{World } A_Z \end{bmatrix}_t \Delta t^2 \quad (157)$$

$$\begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_t + \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix}_t \Delta t \quad (158)$$

$$[\Psi \ \theta \ \phi] = \text{Dead Reckon} \left(\begin{bmatrix} \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix} \right) \quad (159)$$

26.4.11 Algorithm Nine, Constant Body Acceleration with Fixed Axes

Algorithm nine is a great simplification of algorithm eight. The entity can not rotate so the translational velocity and acceleration can be directly transformed into world axes components. The position is dead reckoned with a second order equation. The body axes velocity is dead reckoned with a first order equation. The orientation and the translational acceleration remain constant. The angular velocity is zero.

$$\begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} = \text{Generate Matrix}([\Psi \ \theta \ \phi]) \quad (160)$$

$$\begin{bmatrix} \text{World } A_X \\ \text{World } A_Y \\ \text{World } A_Z \end{bmatrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix} \quad (161)$$

$$\begin{bmatrix} \text{World } V_X \\ \text{World } V_Y \\ \text{World } V_Z \end{bmatrix} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix} \quad (162)$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_t + \begin{bmatrix} \text{World } V_X \\ \text{World } V_Y \\ \text{World } V_Z \end{bmatrix}_t \Delta t + \frac{1}{2} \begin{bmatrix} \text{World } A_X \\ \text{World } A_Y \\ \text{World } A_Z \end{bmatrix} \Delta t^2 \quad (163)$$

$$\begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} V_X \\ V_Y \\ V_Z \end{bmatrix}_t + \begin{bmatrix} A_X \\ A_Y \\ A_Z \end{bmatrix} \Delta t \quad (164)$$

26.5 Special Rotation Algorithms for Dead Reckoning

The following methods are used to manage entity orientation. The first method is used to produce a rotation matrix based on the DIS euler angles. This method is used by algorithm six and nine. This same method is also part of the dead reckoning method used by algorithms three, four, seven, and eight. The rotation matrix is also produced for use in the dead reckoning algorithms. Because of the difficulty in manipulating euler angles, the rotation matrix was chosen as the format to use for dead reckoning. This rotation matrix is then decomposed to generate an equivalent set of euler angles after the dead reckoning. This maintains the orientation in euler angles for the clients while allowing the Entity Service something more useful to work with.

26.5.1 Generation of the Initial Rotation Matrix

The rotation matrix is given by,

$$\text{Orientation} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} = f(\psi, \theta, \phi)_{PDU} \quad (165)$$

The matrix functions as follows,

$$\begin{bmatrix} Q_X \\ Q_Y \\ Q_Z \end{bmatrix}_{Body} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix} \begin{bmatrix} Q_X \\ Q_Y \\ Q_Z \end{bmatrix}_{World} \quad (166)$$

$$\begin{bmatrix} Q_X \\ Q_Y \\ Q_Z \end{bmatrix}_{World} = \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}^T \begin{bmatrix} Q_X \\ Q_Y \\ Q_Z \end{bmatrix}_{Body} \quad (167)$$

And the terms of the matrix are determined by

$$r_{XX} = \cos \psi \cos \theta \quad (168)$$

$$r_{XY} = \sin \psi \cos \theta \quad (169)$$

$$r_{XZ} = -\sin \theta \quad (170)$$

$$r_{YX} = -\sin \psi \cos \phi + \cos \psi \sin \theta \sin \phi \quad (171)$$

$$r_{YY} = \cos \psi \cos \phi + \sin \psi \sin \theta \sin \phi \quad (172)$$

$$r_{YZ} = \cos \theta \sin \phi \quad (173)$$

$$r_{ZX} = \sin \psi \sin \phi + \cos \psi \sin \theta \cos \phi \quad (174)$$

$$r_{ZY} = -\cos \psi \sin \phi + \sin \psi \sin \theta \cos \phi \quad (175)$$

$$r_{ZZ} = \cos \theta \cos \phi \quad (176)$$

These terms are very standard elements of euler angle rotation matrices. They are more useful because they allow for vector components to be directly resolved. Keeping the orientation as angles would require these operations every time a vector needed to be resolved into another frame.

26.5.2 Dead Reckoning of Orientation

In order to dead reckon entity rotation, the form of the angular velocity is resolved from a basic vector into a magnitude and normal vector. This allows for dead reckoning of the entity rotation matrices. Note that the magnitude of the rotation may be determined to be zero, which it should not be in these models. In that case, the normal vector of rotation does not exist and should not be solved for.

$$\text{Magnitude of Rotation} = \Omega = \sqrt{\omega_X^2 + \omega_Y^2 + \omega_Z^2} \quad (177)$$

$$\text{Normal Vector of Rotation} = \begin{bmatrix} n_X \\ n_Y \\ n_Z \end{bmatrix} = \begin{bmatrix} \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix} / \Omega \quad (178)$$

The orientation dead reckoning is performed in three different steps. The first is determining a quaternion from the given rotation and time step. This begins with finding the actual angle of rotation, divided by two to reduce computations. Then find each of the euler parameters.

$$\text{Half Angle of Rotation} = \gamma = \Omega \Delta t \quad (179)$$

$$\text{Euler Zero} = \gamma_0 = \cos (\gamma/2) \quad (180)$$

$$\text{Euler One} = \gamma_1 = n_X \sin (\gamma/2) \quad (181)$$

$$\text{Euler Two} = \gamma_2 = n_Y \sin (\gamma/2) \quad (182)$$

$$\text{Euler Three} = \gamma_3 = n_z \sin(\gamma/2) \quad (183)$$

The next step is to determine the intermediate rotation matrix that is generated for the current time step. It relates the new orientation to the orientation at the beginning of the current time step.

$$\text{Intermediate Matrix of Rotation} = \begin{bmatrix} e_{XX} & e_{XY} & e_{XZ} \\ e_{YX} & e_{YY} & e_{YZ} \\ e_{ZX} & e_{ZY} & e_{ZZ} \end{bmatrix} \quad (184)$$

The relationship is

$$\begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_{t+\Delta t} = \begin{bmatrix} e_{XX} & e_{XY} & e_{XZ} \\ e_{YX} & e_{YY} & e_{YZ} \\ e_{ZX} & e_{ZY} & e_{ZZ} \end{bmatrix} \begin{bmatrix} r_{XX} & r_{XY} & r_{XZ} \\ r_{YX} & r_{YY} & r_{YZ} \\ r_{ZX} & r_{ZY} & r_{ZZ} \end{bmatrix}_t \quad (185)$$

With the elements of the matrix given by

$$e_{XX} = 2(\gamma_0^2 + \gamma_1^2) - 1 \quad (186)$$

$$e_{XY} = 2(\gamma_1\gamma_2 - \gamma_0\gamma_3) \quad (187)$$

$$e_{XZ} = 2(\gamma_1\gamma_3 + \gamma_0\gamma_2) \quad (188)$$

$$e_{YX} = 2(\gamma_1\gamma_2 + \gamma_0\gamma_3) \quad (189)$$

$$e_{YY} = 2(\gamma_0^2 + \gamma_2^2) - 1 \quad (190)$$

$$e_{YZ} = 2(\gamma_2\gamma_3 - \gamma_0\gamma_1) \quad (191)$$

$$e_{ZX} = 2(\gamma_1\gamma_3 - \gamma_0\gamma_2) \quad (192)$$

$$e_{ZY} = 2(\gamma_2\gamma_3 + \gamma_0\gamma_1) \quad (193)$$

$$e_{ZZ} = 2(\gamma_0^2 + \gamma_2^2) - 1 \quad (194)$$

26.5.3 Determining Euler Angles from a Rotation Matrix

The final process is converting the resultant rotation matrix back into euler angles so that the cli-

ents can understand the orientation. This is done using standard methods for rotation matrix decomposition. The decomposition takes places after the dead reckoning of the orientation, or whenever the euler angles are needed.

$$\theta = \text{asin}(r_{XZ}) \quad (195)$$

$$IF\left(\theta = \pm\frac{\pi}{2}\right) \text{ A Singularity Exists} \quad (196)$$

$$ELSE \text{ A Singularity Does Not Exist} \quad (197)$$

If a singularity does not exist, the solution procedure is as follows.

$$\cos\psi = r_{XX}/\cos\theta \quad (198)$$

$$\sin\psi = r_{XY}/\cos\theta \quad (199)$$

$$IF(\sin\psi \geq 0.0) \quad \psi = \text{acos}(\cos\psi) \quad (200)$$

$$IF(\sin\psi < 0.0) \quad \psi = -\text{acos}(\cos\psi) \quad (201)$$

$$\cos\phi = r_{ZZ}/\cos\theta \quad (202)$$

$$\sin\phi = r_{YZ}/\cos\theta \quad (203)$$

$$IF(\sin\phi \geq 0.0) \quad \phi = \text{acos}(\cos\phi) \quad (204)$$

$$IF(\sin\phi < 0.0) \quad \phi = -\text{acos}(\cos\phi) \quad (205)$$

Otherwise, if a singularity does exist, the solution procedure is as follows.

$$\phi = 0.0 \quad (206)$$

$$\cos\psi = r_{YY} \quad (207)$$

$$\sin\psi = -r_{YX} \quad (208)$$

$$IF(\sin\psi \geq 0.0) \quad \psi = \text{acos}(\cos\psi) \quad (209)$$

$$IF(\sin\psi < 0.0) \quad \psi = -\text{acos}(\cos\psi) \quad (210)$$

26.6 Smoothing from Current to New States

As was mentioned before, the Entity Service currently uses three different approaches to smoothing out the differences between current ghosts and new updates from PDUs. The first approach is not really smoothing at all, as state information is snapped from the current state to the new state the instant the new state is received. The other two approaches take some amount of time to converge towards the new data. A goal state is predicted using the new state and dead reckoning algorithm from the PDU. Then the current state is gradually converged into the goal state so that continuity of motion is maintained.

26.6.1 Snapping State Values

Snapping state values is obviously not really a form of smoothing because it is an instantaneous transition. However, it is included because it was so simple to implement and because it can be useful when interacting with other entity sources that also snap between state values. To perform a snap, simply take the state values directly out of the PDU and begin using them immediately. In other words, all prior state information prior to the new PDU came in is forgotten. As far as the state model is concerned, it starts over from scratch with each new PDU, resetting the state values and beginning dead reckoning anew, as if it were the first PDU that had ever come from that entity.

This approach can cause problems with different programs. A model being displayed by an image generator might seem to teleport from one location and orientation to another. Any type of tracking system, such as the ones used by the DTRs to trace vehicle paths, may not be able to cope with the sudden discontinuity and may produce unusable results. All these problems can be fixed if each application smooths out the entity state itself. However, this is an unnecessary and common burden that can easily be moved into the Entity Service by having it perform smoothing when required (or simply desired) by its clients.

26.6.2 Determining a Goal State

Since smoothing takes time to converge to a new state, that new state has to be determined before the smoothing can be setup. This new state, or goal state, is the objective of the smoothing algorithm. In other words, the current state should be smoothed towards the goal state, until it is identical to the goal state, at which point the standard dead reckoning will take over. Whenever a new PDU arrives, the process begins again. Note that significant tolerance error could build up if the ghost does not reach the goal state on a regular basis. For this reason, an average PDU cycle time is maintained as the PDUs arrive. The goal state is determined by using half of this average cycle time, to insure that the goal state is reached in the majority of cases.

Since the dead reckoning algorithms have already been discussed in detail, the explanation of how the goal state is calculated will be limited to a basic overview.

First, two the state and dead reckoning algorithm are read straight out of the PDU and saved independently from the ghost.

$$\text{New State (straight out of PDU)} = S_{PDU} \quad (211)$$

$$\text{Dead Reckoning Algorithm} = dr_{PDU} \quad (212)$$

Next, the average time between PDUs is updated to include the update time that was just observed when the new PDU came in. Note that this average is initialized with fifty update cycles of one second each. Then the goal time is predicted using half of the current average.

$$\text{Average Time Between PDUs} = \Delta t_{Ave} \quad (213)$$

$$\text{Time to Reach Goal} = \Delta t_{Goal} = \frac{\Delta t_{Ave}}{2} \quad (214)$$

Finally, the goal state is predicted using the dead reckoning algorithms discussed in the prior sections. This goal state, along with the time to reach goal, is then used by one of the smoothing algorithms as the ending condition for the convergence.

$$\text{Goal State} = S_{Goal} = \text{deadReckon}(S_{PDU}, dr_{PDU}, \Delta t_{Goal}) \quad (215)$$

26.6.3 Linear Interpolation between States

Interpolation between states is the simplest of the two methods used to smooth the state transitions. By sampling the ghost at the moment a new PDU is received, the current state can be saved. With the goal state computed as described in the prior section, the boundary conditions of the interpolation are set. It is then simply a matter of calculating the differences between the two states and setting up ratios based on the length of time until the goal has to be reached.

Linear interpolation was chosen because it is the simplest form of interpolation to implement. In effect, each state value is independent of all other state values and linear relationships can be setup separately for each state value. Each component of each vector has its own private equation. The sole exception to this approach is the entity's orientation, which will be discussed after basic interpolation.

$$\text{Interpolation Value} = S_N \quad (216)$$

Where S and N are defined as

A State Vector

$$S = (\text{Position, Vecocity, Acceleration,} \quad (217)$$

Angular Velcocity, or Articulation)

$$N = \text{The Vector Component (x,y,or z)} \quad (218)$$

Given a starting point, the current value, and a goal point, the difference between the two is calculated. Any value in-between the two boundary values can be determined by determining the ratio

of the sample time in the interval, multiplying by the difference, and adding back in to the start value.

$$\text{Parameterized Time} = \Delta t_{Ratio} = \frac{\Delta t_{Sample}}{\Delta t_{Interval}} \quad (219)$$

$$\text{State Difference} = \Delta S_N = S_{N, Goal} - S_{N, Start} \quad (220)$$

$$\text{Sampled State} = S_{N, Sample} = S_{N, Start} + \Delta S_N \Delta t_{Ratio} \quad (221)$$

Since orientation is not a vector quantity, the values in an orientation relationship cannot be treated independently. Because the orientation is already expressed in terms of a matrix for use in the dead reckoning equations, the matrix elements are used in the interpolation. The first step is to interpolate the matrix elements the same way all the other values were interpolated. Then the sample matrix is processed to insure that the matrix is still a valid rotation matrix, which it probably won't be at first. One element is selected to remain exactly the same as the interpolated value, namely element r_{XZ} , because of its importance in determining the equivalent euler angles from the matrix. Then the row and column of this element are normalized, by modifying the two remaining elements in each one. Then the remaining four elements are modified by normalizing the remaining two rows and columns.

26.6.4 Fitting a Hermite Spline between States

As an alternate to simple interpolation, splines are also used to smooth the transitions between states. Fitting a spline through the current state and goal state allows for a smoother transition than a straight line interpolation allows. Hermite splines are used because they are one of the most straightforward types of splines and can easily be used to model basic, curved paths. To fit a hermite spline through two states requires only the values and the derivatives of the values at each state. Again, each vector component can be smoothed independently. But since the value and its derivative are both considered, the result is a much smoother transition between states than the simple interpolation model provides. The basis for the hermit spline is a general spline equation incorporating a parameterized time vector, the hermite basis matrix, and the hermite geometry vector, which is made up of the state values and their derivatives at each endpoint. The equation is as follows. Note that the hermite basis matrix is a constant for all hermite splines and is used for every state value.

$$\text{Sampled State} = S_{N, Sample} = \Delta T_H M_H G_H \quad (222)$$

Where the components of the equation are as follows,

$$\text{Hermite Time Vector} = \Delta T_H = [\Delta t^3 \ \Delta t^2 \ \Delta t \ 1] \quad (223)$$

$$\text{Hermite Basis Matrix} = M_H = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad (224)$$

$$\text{Hermite Geometry Vector} = G_H = \begin{bmatrix} S_{N, Start} \\ \dot{S}_{N, Start} \\ S_{N, Goal} \\ \dot{S}_{N, Goal} \end{bmatrix} \quad (225)$$

A hermite spline equation is to all of the vectorial values of the entity. For fitting position values to a spline, the derivatives must be expressed as global velocities. Fitting velocity is done using the acceleration for the derivative as is, since it is guaranteed to be in the same frame. The values of an articulation are fit similarly, using their velocities, if given, as the derivatives. Since acceleration, angular velocity, and articulation velocity are constants at each boundary state, their derivatives are set to zero and their equations are simplified accordingly. Again, orientation is a difficulty, as it is not a vector quantity. Due to time constraints, a spline equation has not yet been made for the orientation. The orientation interpolation approach described in the prior section is used instead.

27.0 IG Host User's Manual

27.1 Introduction to the IG Host User's Manual

DracVision¹ (DV) is a powerful and flexible program to control and instantiate single or multiple image generators (IGs). In particular, the DV host process can instantiate an IRIS Performer IG or communicate with an Evans & Sutherland IG, a modified ESIG2000. Through the creative and innovative efforts of the Dynamic Terrain (DT) and the Dynamic Virtual Environments (DVE) projects in the Visual Systems Lab (VSL) of the Institute for Simulation and Training (IST), DracVision supports various vehicle models and DT within a Distributed Interactive Simulation (DIS) environment.

DIS is handled conveniently through the Entity Service (ES) process, which reads all incoming packet traffic from the DIS network, and offers the capability of writing to the network as well. All DIS v2.0.3 specific Protocol Description Units (PDUs) are recognizable. This PDU is specified in other IST documentation.

Dynamic Terrain is not currently part of the DIS standard. The Terrain Update PDU specified by the DT project is handled by the Terrain Service (TS) process. It is important to note that the terrain update packets travel through a different port than normal DIS traffic. DT packets are generally quite large and exceed the maximum packet size for broadcast ethernet. Thus, they need to be sent out via unicast ethernet links to every machine which wants to receive Dynamic Terrain updates. Using a different port for terrain data also reduces the number of collisions with regular DIS traffic and allows DT specific applications to operate in conjunction with DIS v2.0.3 compliant applications without either crashing due to bogus packets. Of course, the attach PDU is a prototypical packet and it would not be used in a DIS compliant network.

A problem with the Terrain Service must be addressed at this point. In theory, the total shared environment for Dynamic Terrain is inviting. It would allow for multiple DT applications on the same machine. However, in practice, this is not possible for the IG, which **must** run in real-time. It has been shown that the current version of the Terrain Service does not operate well with the continuous terrain level of detail algorithm. Succinctly, the TS updates too slowly for real-time updates. Real-time updates are considered anything over 10Hz, but the TS updates at less than 1Hz. Thus, the current solution is to instantiate the Dynamic Terrain Database within the IG and read terrain changes directly from the network. The updates are much faster and do not significantly slow down the continuous terrain level of detail algorithm, though there will always be room for improvement.

1. DracVision is DT's alias for the IG Host.

27.1.1 System Configuration Requirements

27.1.1.1 Software

- IRIS Performer v1.2 or v2.0, Silicon Graphics, Inc. (SGI)
- C++ v3.2 or v4.0, SGI
- DracVision, IST

27.1.1.2 Hardware

- IRIS 4D Series Workstation or better, SGI
- Ethernet LAN recommended for network support

27.1.1.3 Operating System

- IRIX 4.0.5x
- IRIX 5.3 highly recommended

27.1.2 Project Management

In one way or another, the following projects have sponsored DracVision:

- Dynamic Terrain (DT), US Army STRICOM, IST/VSL
- Dynamic Virtual Environments (DVE), US Army STRICOM, IST/VSL

27.2 Start-up/Configuration

Usage:

host [**@<script.cfg>**] [**@<script.init>**] [**-n<level>**] [**-f<frame_rate>**] [**-h<host>**]

The text window from which DracVision gets executed will be cleared and the following message will be displayed (or one similar to it):

```
IG Host V1.0
Copyright (c) 1994
All Rights Reserved
```

```
The Institute for Simulation and Training
Visual Systems Lab
The University of Central Florida
```

The current version is given on the first line, as well as the version start date and copyright information. After this message, the command prompt will be shown, and the output (if any) from the configuration and initialization files will appear as they are processed. If no prompt is visible after the system has initialized and the IG is running, just press the *return* key a few times until the prompt reappears.

27.2.1 Command Line Parameters

Upon running the host process, the user may specify a script configuration and a script initialization file. A script file is a text file which contains a series of commands. A simple batch file or shell script is another example. A configuration file is a script file which should be read in and executed before initialization. Before the IG process begins initialization (starting Performer and other commands), the configuration file will be executed. After the configuration file is executed and initialization, based upon the configuration, occurs, then the initialization file is read and executed. This file could be considered a batch file. The command line arguments are processed twice. During the first time through, any configuration or initialization file name is copied. Then the configuration file is actually processed before the next command line pass. After the second pass, the host and any IGs are instantiated with the current configuration and then the initialization file is processed. This unique procedure allows for pre-initialization files and subsequent run-time script files. In fact, the initialization file may be processed multiple times, or multiple initialization files may be processed. Furthermore, both script files accept a command starting with the '@' symbol, which specifies another script file. Be careful not to generate infinitely recursive script files! The default file name for the DracVision configuration file is "IGhost.cfg" and the default file name for the DV initialization file is "IGhost.init".

The configuration files are text based, they are not binary. Blank lines are ignored so the user may put them in as often as desired for readability. Also, comments may be included by putting a pound sign, '#', at the beginning of the line. Starting white space is not trimmed. The general format for a line in the file is "**command** *argument1 argument2*"

The "-n" option sets the desired debugging *level*. DracVision will output messages to the window depending on what print mode is desired. The level must be an integer greater than or equal to 0. The value given corresponds to the IRIS Performer debugging level used in the pfNotify() routine. The currently defined levels are:

- PFNFY_ALWAYS 0
- PFNFY_FATAL 1
- PFNFY_WARN 2
- PFNFY_NOTICE 3
- PFNFY_INFO 4
- PFNFY_DEBUG 5
- PFNFY_FP_DEBUG 6
- PFNFY_INTERNAL_DEBUG 7

Furthermore, higher debugging levels may be found by giving a higher number than the constants defined above. However, it is unlikely that such undocumented low levels will exist. For further descriptions of these levels, please refer to the IRIS Performer reference page on pfNotify.

The "-f" option sets the IRIS Performer IG frame rate to the given floating point value in *frame_rate*. The IG will attempt to configure itself to the given frame rate, but if it is unable to do so, then the closest available frame rate will be chosen. The frame rate is specified in frames per second, a common computer graphics term, meaning screen updates per second. The actual rate used depends on the hardware video retrace rate, however. The frame rate must be an integral multiple of the video hardware. For instance, if the refresh rate is 60Hz, then the user may specify

60.0, 30.0, 20.0, 15.0, 12.0, etc. Generally, the video rate is either 60Hz or 72Hz.

The last command line option specifies the type IG host to be used, if only one IG is to be connected or instantiated. It was the original design goal to allow the IG host to control more than one visual image generator. However, implementation was limited later on in the project and currently only one IG can be controlled by the IG host. Only two choices are currently available and *host* must be either **DV** or **ESIG2000**. Note that this option is specified through a character string rather than some constant integer, but upper or lower case is irrelevant.

27.2.2 How to Use Dynamic Terrain

It is sometimes very important to turn DT on or off between executions of DracVision. Running with DT can use much of the CPU if other things must be tested and the terrain aspect of the simulation is not necessary. DracVision, like any IG, can load in static terrain data files if DT is not being used. These files may include references to features or they may be specified independently. Features are objects such as buildings, trees, and roads.

The three most important commands which will affect the operation of DT are *dt*, *dttype*, and *dtload*. *dt* is executed from the IG host configuration file and tells the IG Host whether it should instantiate a Terrain Service Client Interface and attach to the Terrain Service. *dttype* is executed from the IG configuration file and tells the IG what type of terrain update object it should implement for processing changes in the dynamic terrain. *dtload* is executed from either the IG host initialization file (as an argument to *command*) or from the IG command line during run-time and loads the (possibly large) Dynamic Terrain Database, if the *dttype* should happen to "dtdb." If "*dt on*" is a line in the IG host configuration file, then no DT can occur in the IG unless the DT type is "dsci". This particular command refers to the attachment of the IG host to the Terrain Service for terrain clamping of entities (useful for last minute database and entity correlation). So, if the IG host attaches to the Terrain Service (obviously meaning that the Terrain Service is running), then the IG cannot bind to the same port and the DTDB type cannot run properly. Thus, to use the faster version of DT in the IG, set *dt* to *off*.

Should the IG attempt to attach to the Terrain Service and the service is not running, the program will halt and the window may be logged out. This occurs due to an extra *exit()* call in the client interface. In a future release, this problem will not occur, but it may be easily avoided by either not attempting to attach to the Terrain Service or by making sure that it is running correctly.

dttype should be either "dsci", "dtdb", or "none" for no Dynamic Terrain. With "none," all subsequent DT commands will be ignored, and no DT is possible after the configuration is complete. To use the Terrain Service, set the *dttype* to "dsci," which means Database Service Client Interface. Although this type fits in the shared environment architecture, it is currently too slow for a real-time IG. The other type, "dtdb", is much faster and also supports the network DT PDU output by other Terrain Service processes on other machines. "dtdb" means that the Dynamic Terrain Database is instantiated directly without having to go through the service. However, no other applications which need to bind to the network port may be run, and this includes the IG host.

Finally, *dtload* loads the DTDB when the DT type is "dtdb." This command and the other IG DT

commands are not executed if the DT type is "dsci" or "none." It is highly recommended that the user specify the database file, port, and continuous terrain level of detail settings before the call to *dtload*. As most data files are quite large, this command usually takes a few minutes to complete. For more information on how the DTDB loads the data file, please refer to the appropriate section.

27.2.3 IG Host Configuration File (IGhost.cfg)

27.2.3.1 framerate

Usage: `framerate [n]`

Sets the framerate to *n*. This command can only be executed through the configuration file. If you wish to know the current framerate, please see the command "print."

27.2.3.2 ekey

Usage: `ekey [n]`

Sets the entity services shared memory key to *n*. This may be changed only within the configuration file as the IG host attaches to the entity services only once, during initialization. For more information on shared memory keys see the document on the Entity Service.

27.2.3.3 dkey

Usage: `dkey [n]`

Sets the terrain services shared memory key to *n*. This may be changed only within the configuration file as the IG host attaches to the terrain services only once, during initialization. For more information on shared memory keys see the document on the Terrain Service.

27.2.3.4 host

Usage: `host [BV | ESIG2000]`

From within the configuration file, you may select the IG that the IG host will control. Only two image generators are possible from this selection: BwanaVision and the ESIG 2000.

27.2.3.5 debug

Usage: `debug [x]`

Changes the debug warning level to the integer *x* (range 1-5) or you may also use the word which signifies the debug level. They are:

- fatal(1) - Fatal error messages; dying, dead, or gone
- warn(2) - Serious warning messages; may be best to restart
- notice(3) - Warning; up to you
- info(4) - Informational message to the user

- debug(5) - Informational message to the programmer

27.2.3.6 dt

Usage: dt [on | off]

Turns the Dynamic Terrain on or off. If off, then the IG host will not connect to the Terrain Service (which would allow it to run on machines which do not have enough shared memory segments).

27.2.3.7 em

Usage: em [on | off]

Turns the handling of DIS entities, fires, explosions, and other PDU information on or off. If off, then the IG host will not connect to the Entity Service. However, this makes the application host run much faster than the IG (draw process) and thus causes queue filling errors. The host attempts to send too many view point control messages to the IG, which cannot keep up.

27.2.4 DracVision IG Configuration File (IG.cfg)

27.2.4.1 mp

Usage: mp [mode]

Set the multiprocessing mode. The choices are:

- default - depends on the capabilities of the system
- appculldraw - do not fork
- appcull_dl_draw - do not fork, sharing the same display list
- appcull_draw - forked draw
- app_culldraw - forked cull with draw
- app_cull_dl_draw - forked cull and draw, sharing the same display list
- app_cull_draw - forked cull and draw
- appcullodraw - forked draw, with overlapping
- app_cullodraw - forked cull and draw, with overlapping

Please see the Performer man pages for more details on multiprocessing (e.g., "man pfMultiprocess").

27.2.4.2 input

Usage: input [X | GL]

Set the input mode to use either GL queues and windows or GLX windows and asynchronous X event handlers. The GL input scheme will not be supported in OpenGL. Furthermore, because the X event handler runs asynchronously, the draw process will be quicker than for GL input, but this will only be noticeable during large amounts of input.

27.2.4.3 dttype

Usage: `dttype [none | dtddb | dsci]`

Set the type of Dynamic Terrain processing to be used by the IG. The three choices are shown in the usage statement. With "none," all subsequent DT commands will be ignored, and no DT is possible after the configuration is complete. To use the Terrain Service, set the dttype to "dsci," which means Database Service Client Interface. Although this type fits in the shared environment architecture, it is currently too slow for a real-time IG. The other type, "dtddb," is much faster and also supports the network DT PDU output by other Terrain Service processes on other machines. "dtddb" means that the Dynamic Terrain Database is instantiated directly without having to go through the service. However, no other applications which need to bind to the network port may be run.

27.3 Run-Time Commands

27.3.1 IG Host Initialization File (IGhost.init)

27.3.1.1 clamp

Usage: `clamp [on | off]`

Turns entity terrain clamping on or off. If terrain clamping is on then all land based entities will be clamped (projected) onto the terrain as seen by the terrain service. However, the pitch and roll of the entity is NOT changed.

27.3.1.2 filter

Usage: `filter [id | type | name]`

Filter entities by entity ID, type, or by domain. The entity ID is composed of the site, host, and entity number (all integer). The type is defined by kind, domain, country, category, subcategory, specific, and extra (also all integer). For both ID and type, do not separate the fields with commas. For example, to filter out F18 fighter planes, use:

```
filter 1 2 225 1 15 0 0
```

You may also specifically filter out a domain of entities. You may filter "land," "air," "sea," "space," and "subsurface" entities. Please refer to the DIS enumeration standards for specific information on entity enumeration values.

27.3.1.3 unfilter

Usage: `unfilter [id | type | name]`

Unfilter entities by entity ID, type, or by domain. The entity ID is composed of the site, host, and entity number (all integer). The type is defined by kind, domain,

country, category, subcategory, specific, and extra (also all integer). For both ID and type, do not separate the fields with commas. For example, to unfilter F18 fighter planes, use:

```
unfilter 1 2 225 1 15 0 0
```

You may also specifically unfilter a domain of entities. You may unfilter “land,” “air,” “sea,” “space,” and “subsurface” entities which were previously filtered by a filter command. Please refer to the DIS enumeration standards for specific information on entity enumeration values.

27.3.1.4 attach

Usage: attach [site host entity] <view port>

Attempt to attach the IG ownership to the DIS entity specified by the ID (site host entity). If it is not possible, the current ownership does not get reset. A view port may also be specified with the string defining the view port. There is always a “Standard” view port. Others may be found through the “listmapfull” command or by looking in the view point configuration file. You may use print to show the ownership.

27.3.1.5 unattach

Usage: unattach

Unattach from the current ownership. You may use attach or print to display the current ownership.

27.3.1.6 vp

Usage: vp [view port]

Change the view port to the one specified as a parameter. If it is not there, then no view port offset will occur. The values for view port depend on what is in the view point configuration file read in by the view point processing object. The value “Standard” should always be defined, but others may include: “Commander”, “Gunner”, and “Driver.”

27.3.1.7 vptype

Usage: vptype [type]

Set the view point processing type. The choices are:

- stealth - operate the IG as a stealth, getting all input from either the host application or the graphical user interface (GUI)
- IGstealth - let the IG operate as a stealth on its own and no longer send view point processing packets to the IG
- tether - attach to an entity, but always stay behind it
- orbit - attach to an entity, but orbit around it

- none - do not process the view point at all (don't change it)

27.3.1.8 det

Usage: det [x y z]

Sends a detonation to the IG. This detonation is currently defined as one from a TOW missile. Do not expect any results to show up in the simulation or be experienced by anyone else on the network.

27.3.1.9 command

Usage: command [cmd]

Sends the ASCII command "cmd" to the IG. Obviously, this must be in a format recognized by the IG. You do not need to surround "cmd" in quotes. Anything after the word "command" will be sent as is to the IG. An example command to get help for BV is "command help".

27.3.1.10 fog

Usage: fog [on | off]

Enable or disable the fog.

27.3.1.11 trans

Usage: trans [x y z]

Translate the view point to the absolute coordinates given by [x, y, z]. These coordinates are in meters. The IG host will then be thrust into stealth mode.

27.3.1.12 rot

Usage: rot [h p r]

Rotate the view point to the absolute orientation given by [h, p, r]. These coordinates are in degrees. The IG host will then be thrust into stealth mode. "h" is the heading, "p" is the pitch, and "r" is the roll.

27.3.1.13 transrel

Usage: transrel [x y z]

Translate the view point by the relative coordinates given by [x, y, z]. These coordinates are in meters. The IG host will then be thrust into stealth mode.

27.3.1.14 rotrel

Usage: rotrel [x y z]

Rotate the view point by the relative orientation given by [h, p, r]. These coordinates are in degrees. The IG host will then be thrust into stealth mode. H is the heading, p is the pitch, and r is the roll.

27.3.2 Other IG Host Commands

27.3.2.1 quit

Usage: quit

Requests a clean exit from the IG host. All spawned processes will be killed properly and all memory will be released.

27.3.2.2 help

Usage: help <topic>

The help command gives you the complete listing of textual commands for the IG host or it can give you help with a specific topic, such as this screen. You can only get specific help on one topic at a time.

For any of the commands, arguments to the command will be shown to the right. If the argument is enclosed within square brackets [], then the argument is necessary. If they are shown within angle brackets <>, then the argument(s) are optional. Floating point arguments will be noted by n.n rather than just n. If the argument represents a choice within an enumeration, then the choices will be listed with a logical or | between them.

27.3.2.3 debug

Usage: debug [x]

Changes the debug warning level to the integer x (range 1-5) or you may also use the word which signifies the debug level. They are:

- fatal(1) - Fatal error messages
- warn(2) - Serious warning messages; may be best to restart
- notice(3) - Warning
- info(4) - Informational message to the user
- debug(5) - Informational message to the programmer

27.3.2.4 print

Usage: print <x>

Displays the current status of the variable x. The possible choices for x are:

- framerate - user defined framerate, not the current framerate
- host - which IG, if any, are we using
- vp - current view point
- ekey - entity service shared memory key
- dkey - terrain service shared memory key
- ownship - if the host has an ownship entity, print out its information
- debug - the current debugging level

If x is not given, then the status of all variables are printed. You may abbreviate print with the single letter "p".

27.3.2.5 (repeat command)

Usage: ,,

Repeat the previous command. The system remembers only the most recent command (or last command in a configuration file). You may also use !! instead of the two commas.

Usage: !!

Repeat the previous command. The system remembers only the most recent command (or last command in a configuration file). You may also use ,, instead of the two commas.

27.3.2.6 list

Usage: list

Lists all the entities from the entity state processing object in a concise format. Only the entity ID, entity type, IG bind ID, and current list status is shown. For a more detailed listing of entities, please see the command "listfull."

27.3.2.7 listfull

Usage: listfull

Lists all the entities from the entity state processing object in a fully expanded format. All information on each of the entities is displayed. If there are many entities, this could be quite long. For a much more concise listing, please see the command "list."

27.3.2.8 listbind

Usage: listbind

Lists the IDs of all entities bound to the IG. This number is used between the IG host and the IG to identify the entities.

27.3.3 Other DracVision Commands

27.3.3.1 quit

Usage: quit

Requests a clean exit from the IG. All spawned processes will be killed properly and all memory will be released.

27.3.3.2 help

Usage: help <topic>

The help command gives you the complete listing of textual commands for the IG or it can give you help on a specific topic, such as this screen. You can only get specific help on one topic at a time.

For any of the commands, arguments to the command will be shown to the right. If the argument is enclosed within square brackets [], then the argument is necessary. If they are shown within angle brackets <>, then the argument(s) are optional. If the argument represents a choice within an enumeration, then the choices will be listed with a logical or | between them.

27.3.3.3 print

Usage: print <x>

Displays the current status of the variable x. The possible choices for x are:

- fade - fade LOD parameters
- fov - field of view parameters
- clip - clipping plane information
- esky - information on the earth sky model, including moon and fog
- models - static model manager
- vp - current view point and orientation
- mode - viewing mode: stealth or other
- lighting - true or false
- texture - true or false
- wireframe - true or false

If x is not given, then the status of all variables are printed. You may abbreviate print with the single letter p (as in dbx).

27.3.3.4 listmap

Usage: listmap

Lists all the mapping elements used in the mapping of entity types to the hierarchical object class structure.

27.3.3.5 listmapfull

Usage: listmapfull

Lists all the mapping elements used in the mapping of entity types to the hierarchical object class structure. This command provides a much more detailed output than listmap.

27.3.3.6 listgeom

Usage: listgeom

Lists all the elements in the geometry generator.

27.3.3.7 listgeomfull

Usage: listgeomfull

Lists all the elements in the geometry generator with a verbose listing.

27.3.3.8 listdets

Usage: listdets

Prints out any ongoing explosions.

27.3.3.9 framerate

Usage: framerate [n]

Sets the framerate to n. The IG will attempt to match the given framerate. The actual framerate set will be an integral multiple of the monitor refresh rate.

27.3.3.10 profile

Usage: profile [on | off]

If the argument is "on" then profiling channel statistics will be displayed at the top of the window. This is a feature offered through the Performer library.

27.3.3.11 window

Usage: window [left right bottom top]

Sets the position of the IG window to the pixel specifications of the arguments [left right bottom top].

27.3.3.12 wireframe

Usage: wireframe [on | off]

Sets the wireframe viewing mode of the IG.

27.3.3.13 lighting

Usage: lighting [on | off]

Usually lighting (and thus shading) is enabled, but you may disable the lighting by issuing "lighting off."

27.3.3.14 texture

Usage: texture [on | off]

To turn texturing on, this command is used. Though it should work with Indigo machines, it is not recommended. It may drop the framerate to insufficient levels for real-time operation (e.g., 0.2 Hz).

27.3.3.15 antialias

Usage: antialias [on | off]

To turn antialiasing on, this command is used. This command will not work on any machine except one that supports antialiasing in hardware, such as the Reality Engine.

27.3.3.16 fovx

Usage: fovx [x]

Set the field of view in the X direction. This is a floating point parameter which specifies the angle of the field of view in degrees.

27.3.3.17 fovy

Usage: fovy [y]

Set the field of view in the Y direction. This is a floating point parameter which specifies the angle of the field of view in degrees.

27.3.3.18 clipnear

Usage: clipnear [near]

Set the distance to the near clipping plane in floating point meters.

27.3.3.19 clipfar

Usage: clipfar [far]

Set the distance to the far clipping plane in floating point meters.

27.3.3.20 overlay

Usage: overlay [on | off]

Toggle the overlay plane drawing. This toggles the program ID in the upper right corner. You can also use the "O" key.

27.3.3.21 tod

Usage: tod [n | h m | h:m]

Assign the current time of day. By default, the IG gets the time of day from the system timer. This command should be used if this time is incorrect or a different time of day is desired. Currently, time is not passed to the IG through the network. n specifies a floating point value in the range 0.0 to 1.0 which corresponds to mid-night to noon. From noon to midnight is merely the reverse (1.0 to 0.0). The time of day is currently scaled linearly. To have the IG recompute the time of day from the system clock, use "tod -1". You may also specify the time of day via two parameters: hours and minutes. These are integer values. Hours range in military time from 0 to 23 and minutes range from 0 to 59. Similarly, a colon may inserted between the hours and minutes to improve readability.

27.3.3.22 stars

Usage: stars [on | off]

Toggle the stars in the sky. Drawing the stars takes from 2 ms (on an Onyx-4 Reality Engine 2) to about 5 or 6 ms (on an Indigo₂ Extreme).

27.3.3.23 starfield

Usage: starfield [r p t n]

Set the star field with radius *r*, number of divisions in the phi direction *p*, number of divisions in the theta direction *t*, and the maximum number of stars *n*. The stars are positioned in random patterns distributed evenly throughout the sky. They will have a grey color between 0.7 and 1.0 (where 1.0 is totally white and 0.0 is totally black). Also, they will be slightly transparent with an alpha value between 0.0 and 0.3, where 1.0 is totally transparent.

27.3.3.24 stealth

Usage: stealth [on | off]

Implement stealth in the IG and override any view point commands coming in from the IG host. The mouse controller has been defined especially for easy movement about complicated objects. The following defines how movement commands are interpreted:

- left mouse button: pitch, yaw and forward movement
- middle mouse button: pitch, yaw and backward movement
- right mouse button: roll

- left mouse button with control key pressed: upward movement only
- middle mouse button with control key pressed: downward movement only

- left mouse button with alt key pressed: pitch and yaw only
- middle mouse button with alt key pressed: pitch and yaw only

- for faster forward or backward movement: press the left shift key
- for slower forward or backward movement: press the right shift key

However, Performer left out control for the alt key, so you cannot use it for control when this command has been activated. Problems may also be experienced with the mouse and keyboard control when X input is used, rather than GL input. Specifically, control, shift, and alt commands are not interpreted the same. Until this problem is fixed in Performer, X input will not work correctly.

27.3.3.25 speed

Usage: speed [n]

Set the mouse speed to "n". This value ranges from 0.0 to infinity. It is highly undesirable to set the speed too high, but it will probably depend upon the size of your database and how quickly you want to be able to move through. Of course, this command will only affect mouse control when you are in the IG stealth mode.

27.3.3.26 sens

Usage: sens [n]

Set the mouse sensitivity to "n". This value ranges from 1 to 10. The higher the value of "n", the more sensitive the mouse becomes. This will result in quicker turns. Of course, this command will only affect mouse control when in the IG stealth mode.

27.3.3.27 fog

Usage: fog [on | off]

Enable or disable the fog.

27.3.3.28 fogopaque

Usage: fogopaque [near far]

Set the near and far distances at which fog will become opaque. The near value determines where fog will start to appear and the far value determines where you can see nothing but fog. Both values are floating point. The default opaque range is 1,000.0 for the near value and 35,000.0 for the far value.

27.3.3.29 fogoffset

Usage: fogoffset [near far]

These are merely offsets to the fog opaque values. The default opaque range is 10.0 for the near value and 100.0 for the far value.

27.3.3.30 fogcolor

Usage: fogcolor [red green blue]

Set the RGB color of the fog itself. This is useful for having dark or light fog or even color-diffused fog. Red, green, and blue are floating numbers which should range from 0.0 to 1.0.

27.3.3.31 fogtype

Usage: fogtype [str]

Set the type of fog to be displayed. The possible types are:

- vtxlin - vertex linear, eye space distances
- vtxexp - vertex exponential, for heavy fog
- vtxexp2 - vertex exponential squared, for distant haze
- pixlin - pixel linear, eye space distances
- pixexp - pixel exponential, for heavy fog
- pixexp2 - pixel exponential squared, for distant haze
- spline - pixel spline, spline based fog (only on RE)

27.3.3.32 stress

Usage: stress [on | off]

Toggle the dynamic load management on or off. Dynamic load management tracks the graphics load in the system and modifies a stress variable. This stress is multiplied to the LOD range of every model in the system, effectively pushing out levels of detail when there is too much stress. The goal is obviously to maintain the desired frame rate with a reduced picture quality.

27.3.3.33 stresslow

Usage: stresslow [n]

Sets the low boundary of the load management hysteresis band. This value should be between 0.0 and 1.0, but always less than stresshigh. When the load is between the low boundary and the high boundary, there is no stress on the system (thus stress = 1.0 and no changes in LOD ranges occur).

27.3.3.34 stresshigh

Usage: stresshigh [n]

Sets the high boundary of the load management hysteresis band. This value should be between 0.0 and 1.0, but always greater than stresslow. When the load is between the low boundary and the high boundary, there is no stress on the system (thus stress = 1.0 and no changes in LOD ranges occur).

27.3.3.35 stressscale

Usage: stressscale [n]

Sets the stress scale factor. This is what the current stress value is multiplied by to get a new stress value every frame that the graphics load is greater than the high boundary.

27.3.3.36 stressmax

Usage: stressmax [n]

The stress level is clamped between 1.0 and n. n should never be below 1.0.

27.3.3.37 stresspix

Usage: stresspix [n]

As the levels of detail on the models are affected by the dynamic load management, some large models do not get pushed out. If the projected pixel size of the model is less than n pixels (integer), then it will be affected by the stress level. If it is n pixels or larger then it will not be affected. Fade LOD is automatically implemented for Reality Engines, which support the multisample buffer.

27.3.3.38 fade

Usage: fade [on | off]

Turn the fade level of detail on or off. Fade LOD allows the visual models to become less detailed as they move further away from the view point.

27.3.3.39 fadedist

Usage: fadedist [x]

Set the fade LOD distance to the floating point value x in meters.

27.3.3.40 fadescale

Usage: fadescale [x]

Set the fade LOD scale value to the floating point value x.

27.3.3.41 loadscs

Usage: loadscs [filename x y z h p r]

Load a static model and generate a static coordinate system (SCS) matrix which will position and orient the model correctly in the database. The position is given by [x,y,z] and is in floating point meters. The orientation is specified by [h,p,r] and is given in floating point degrees. The filename specifies a file which can be loaded by Performer. Of course, the types of files that fall into this category depend upon which version of Performer you running. An example type is MultiGen Flight.

27.3.3.42 loaddds

Usage: loaddds [filename x y z h p r]

Load a dynamic model and generate a dynamic coordinate system (DCS) matrix which will position and orient the model correctly in the database. The position is given by [x,y,z] and is in floating point meters. The orientation is specified by [h,p,r] and is given in floating point degrees. The filename specifies a file which can be loaded by Performer. Of course, the types of files that fall into this category depend upon which version of Performer you running. An example type is MultiGen Flight.

27.3.3.43 listmodels

Usage: listmodels

Print out information on all loaded static and dynamic models. These do not include any entities currently bound from the network.

27.3.3.44 listscs

Usage: listscs

Print out information on all loaded static models.

27.3.3.45 listdcs

Usage: listdcs

Print out information on all loaded dynamic models.

27.3.3.46 resetmodels

Usage: resetmodels

Delete all loaded models and remove them from the geometry scene. These do not include any entities currently bound from the network.

27.3.3.47 transdcs

Usage: transdcs [# x y z]

Translate the DCS to the given absolute coordinates. You can find the specific # for a DCS by using the listdcs command.

27.3.3.48 rot dcs

Usage: rot dcs [# h p r]

Rotate the DCS to the given absolute angles. You can find the specific # for a DCS by using the listdcs command.

27.3.3.49 xformdcs

Usage: xformdcs [# x y z h p r]

Transform the DCS to both the given absolute coordinates and the given absolute angles. You can find the specific # for a DCS by using the listdcs command.

27.3.3.50 remdcs

Usage: remdcs [#]

Remove the DCS denoted by #. They will be numbered from 0 to n. You can find the specific # for a DCS by using the listdcs command.

27.3.3.51 remscs

Usage: remscs [#]

Remove the SCS denoted by #. They will be numbered from 0 to n. You can find the specific # for a SCS by using the listscs command.

27.3.3.52 load

Usage: load <filename>

Load all the models defined in the DIS geometry file. This is good for obtaining the best speed while running. It is extremely slow to load a model while in the middle of the simulation. The loader is not asynchronous to the simulation. This command needs only be performed once. If you provide a filename, then only that

file will be loaded. This way you can selectively load only the models you will use without a huge start-up delay. Of course, even without the use of this command, the IG will still load files it might need during run-time. A call to load, with no filename, will guarantee no delay.

27.3.3.53 dtload

Usage: dtload

Once the DT is configured properly, it must be loaded. When using the DTDB option, loading may take a few minutes. No process is already running which has the terrain loaded. However, if the Terrain Service DT option is being used, then the terrain has already been loaded and this command does not take very long. Once the terrain is loaded, then it cannot be unloaded and no other DT commands will be executed.

27.3.3.54 dtport

Usage: dtport [port]

When using the DTDB option, the port must be given. The default port number is 4000. This command is executed, but it is meaningless if the user wants to use the Terrain Service.

27.3.3.55 dtkey

Usage: dtkey [key]

The unique structure of the IG Dynamic Terrain process requires that the main IG application process communicate with a highly specialized IRIS Performer generation process via shared memory. The generation process is forked, or started, and is a child of the IG. Shared memory communication generally requires that a particular "key" be used as the starting place for the shared memory segment and semaphores. The user may change the default key used by the Dynamic Terrain generation process with the dtkey command. The default key is 0x70.

27.3.3.56 dtfile

Usage: dtfile <filename>

The Dynamic Terrain Database (DTDB) file may be specified with this command. The file must be a valid DTDB configuration file, otherwise unexpected results may occur. For information regarding the format of this file, please refer to the DTDB documentation.

27.3.3.57 dtspacing

Usage: dtspacing [n]

The initial spacing of the interior terrain grid is given as an argument to the dtspacing command. The spacing may be a floating point value, but undefined results may occur if it is not a power of two. The units of the spacing are generally consid-

ered to be meters, but in actuality, no units are ever specified in the DTDB. The default value for the spacing is 1.0. The spacing doubles for each boundary.

27.3.3.58 dtrange

Usage: dtrange [n]

The range determines how many squares, or grid blocks, are in each direction of the interior terrain grid. The default value for the range is 16. This range means that the interior grid will be 16 units to the positive and negative X and Y directions, where the units are defined by the spacing parameter. Thus, the interior grid, by default, has an area of 32x32 grid blocks. The next boundary has 3/2 grid blocks of the interior, because the spacing doubles, but the boundary extends range units from the interior.

27.3.3.59 dtlod

Usage: dtlod [n]

The number of levels of detail is given by the parameter to dtlod. With 0 or 1, only the interior grid is visible (actually, 0 is meaningless, but the IG assumes the user meant 1; otherwise dtload would never be called). For each value beyond 1, another level of detail is added and the visible terrain patch gets 3/2 times larger.

27.3.4 DracVision Keyboard Commands

Some commands may be executed by pressing a single keyboard key while the mouse pointer is over the IG window. This is especially useful if the user wishes to keep the window at full screen. The following list briefly describes what keys may be used and what occurs when they are pressed.

- <Esc> - same functionality is *quit*.
- <F,f> - toggles the fog.
- <L,l> - toggles lighting.
- <T,t> - toggles texturing.
- <O,o> - toggles the overlay plane message "DracVision" in the upper right corner.
- <W,w> - toggles wireframe mode.
- <A,a> - toggles antialiasing.
- <F2> - toggles the statistical profiling display. This display possesses interesting information and is fully described in the IRIS Performer reference pages.
- <F3> - toggles between the four frame rate phase locking mechanisms: locked to the frame rate, floating, free (as fast as possible), and limited to the frame rate.
- <F11> - pushes the IG window.
- <F12> - pops the IG window.

References

1. Alluisi, E. "The Development of Technology for Collective Training: SIMNET, a Case History", *Human Factors*, vol. 33, No.3, June 1991
2. Altman, Marty, Kilby, Mark, Lisle, Curtis, "On a Shared Environment Concept for Distributed Simulation", 11th DIS Workshop, Orlando, FL, 1994.
3. BBN Systems and Technologies Corporation, "SIMNET M1 Abrams Main Battle Tank Simulation - Software Description and Documentation (Revision 1)," Report No. 6323, August 1988.
4. Baladi, George Y., Behzad Rohani, "A Mathematical Model Of Terrain-Vehicle Interaction For Predicting The Steering Performance Of Track-Laying Vehicles," Proc. 6th Conf. of the ISTVS, Vol. I, 1978.
5. Balovnev, V.I. *New Methods for Calculating Resistance to Cutting of Soil*. Translated from Russian, Published for the U.S. Department of Agriculture and the National Science Foundation. Washington, D.C., 1983.
6. Bekker, M.G., "Introduction to Terrain-Vehicle Systems," University of Michigan Press, 1969.
7. Bekker, M.G., "Off-The-Road Locomotion," Research and Development in Terramechanics, 1960.
8. Bekker, M.G., "Theory of Land Locomotion: The Mechanics of Vehicle Mobility," University of Michigan Press, 1956.
9. Bromhead, E. N. *The Stability of Slopes*. Surrey University Press, 1986.
10. Campbell, C., "Fluid Dynamics Methodologies for Computer Graphics", *Master Thesis*, Computer Science Department, University of Central Florida, Dec. 1991.
11. Cernica, J. N., *Geotechnical Engineering*. Holt, Rinehart & winston, 1982.
12. Chen, Jim X. and Michelle Sartor. "An Approach to Implementing Fluids on Dynamic Terrain." 12th DIS Workshop, Orlando, Florida, March 1995 (Postscript).
13. Chen, Jim and Michelle Sartor. "Fluids in a Distributed Interactive Simulation." Fifth Annual Conference on AI, Simulation and Planning in High Autonomy Systems, Gainesville, Florida, December 1994.
14. Chen, Jim X. and Niels da Vitoria Lobo. "Towards Interactive-rate Simulation of Fluids with Moving Obstacles by Navier-Stokes Equations." Computer Vision, Graphics and Image Processing (CVGIP): Graphical Models and Image Processing, March 1995 (Postscript).

15. Chen, Jim X.. "Physically-Based Modeling and Real-Time Simulation of Fluids", Ph.D. Dissertation, University of Central Florida, Orlando, Florida, May 1995 (Postscript).
16. Chowdhury, R. N., *Slope Analysis*. Elsevier North-Holland Inc., 1978.
17. Das, Braja M. *Principles of Geotechnical Engineering*. Second Edition, PWS-KENT Publishing Company, 1990.
18. Fournier, A. and Reeves, W. T., "A Simple Model of Ocean Waves", *Computer Graphics* 20:4, pp.75-84, Aug. 1986.
19. Fox, W. R. and MD, A. T., *Introduction to Fluid Mechanics*, John Wiley & Sons Inc., 1992.
20. Gillespie, Thomas D., "Fundamentals of Vehicle Dynamics," SAE 1992.
21. Goss, M. E., "A Real-time Particle System for Display of Ship Wakes", *IEEE Computer Graphics and Applications*, 10:3, pp.30-35, May 1990.
22. Greenwood, Donald T. *Principles of Dynamics*, Second Edition, Prentice-Hall, Inc., 1988.
23. Howe, R.M., "Dynamics of Real-time Digital Simulation," Course notes from Ground Vehicle Dynamics Short Course at Institute for Simulation and Training, Orlando, November 1994.
24. Huang, Y. H., *Stability Analysis of Earth Slopes*. Van Nostrand Reinhold Co., 1983.
25. IST, *The DIS Vision: A Map to the Future of Distributed Simulation*, version 1, Univ. of Central Florida, Orlando, Florida, May 1994.
26. IST, "The DIS Vision," Institute for Simulation and Training, University of Central Florida, IST-SP-94-01, May 1994.
27. IST, "Mobility Across Dynamic Terrain," Institute for Simulation and Training, University of Central Florida, Orlando, 1993.
28. Johnson, R. S., "The SIMNET Visual System", *Proceedings of the Ninth Interservice Training Equipment Conference*, Washington, D.C. Nov. 1987.
29. Karafiath, Leslie L., "Analytical Model For The Turning of Tracked Vehicles In Soft Soils," Proc. of 7th Conf. of the STVS, 1979.
30. Karafiath, Leslie L., "Soil Mechanics for Off-Road Vehicle Engineering," Trans Tech Publications, 1978.
31. Kass, M. and Miller, G., "Rapid, Stable Fluid Dynamics for Computer Graphics", *Computer Graphics* 24:4, pp.49-55, Aug. 1990.

32. Kilby, Mark, Lisle, Curtis, Altman, Marty, Sartor, Michelle, "Dynamic Environment Simulation with DIS Technology", IITSEC Conference Proceedings, Orlando, FL, 1994.
33. Kitano, M., H. Jyozaki, M. Kuma, "An Analysis of Steerability of Tracked Vehicles," Proc. 6th Conf. of the ISTVS, 1978.
34. Kitano, M., H. Jyozaki, "A Theoretical Analysis of Steerability of Tracked Vehicles," Journal of Terramechanics, Vol. 13, No. 4, 1976.
35. Li, Xin. Physically-Based Soil Models of Dynamic Terrain in Virtual Environments. *Technical Report*. CS-TR-92-26, University of Central Florida. Nov. 1992.
36. Li, Xin, and Moshell, J. Michael, "Modeling Soil: Realtime Dynamic Models for Soil Slippage and Manipulation", SIGGRAPH '93 Conference Proceedings
37. Li, Xin, "Physically-based Modelling and Distributed Computation for Simulation of Dynamic Terrain in Virtual Environments," Ph.D. Dissertation, Computer Science Dept., Univ. of Central Florida, May 1993.
38. Lin, Kuo-Chi, "Linear and Non-Linear Models for Tracked Vehicles Simulation," Final Report for the Mathematical Test Bed for System Controller Development, Institute for Simulation and Training, Orlando, IST-CR-93-37, September 93.
39. Lisle, Curtis, Altman, Marty, Kilby, Marty, Sartor, Michelle, "Architectures for Dynamic Terrain and Dynamic Environments in Distributed Interactive Simulation", 10th DIS Workshop, Orlando, FL, 1993.
40. Livshiz, M., Sanvido D.J., Stiles S.D., "Nonlinear Engine Model For Idle Speed Control," Proc. of 33rd IEEE Conference on Decision and Control, Dec. 1994.
41. Lucas, G.G., "Road Vehicle Performance," Vol.7, Gordon and Breach Science Publishers, 1986.
42. Lynn, Martin A., "An Evaluation of Terrain Requirements For Mobility Models," Proc. 4th Conference of the International Society of Terrain Vehicle Systems (ISTVS), Vol. I, 1972.
43. Merritt, H.E., "The Evolution of Tank Transmission," Proc. Institution of Mechanical Engineers, Vol. 154, April 1946.
44. Miller, G. and Pearce, A., "Globular Dynamics: A Connected Particle System for Animating Viscous Fluids", *Computer and Graphics*, 13:3, pp.305-309, 1989.
45. Murakami, H., K. Watanabe, M. Kitano, "A Mathematical Model for Spatial Motion of Tracked Vehicles on Soft Ground," Proc. 10th Conf. of the ISTVS, 1990.
46. Overmars, Mark H., *Forms Library: A Graphical User Interface Toolkit for Silicon Graphics*

Workstations, Utrecht University, 1993.

47. Peachey, D. R., "Modeling Waves and Surf", *Computer Graphics* 20:4, pp.65-74, 1986.
48. Peitgen, Heinz-Otto and Dietmar Saupe, editors. *The Science of Fractals*. Springer-Verlag, New York, New York, 1988.
49. Peyret, R. and Taylor, T. D., *Computational Methods for Fluid Flow*, Springer-Verlag New York Inc. 1985.
50. Potter, M. C. and Wiggert, D. C., *Mechanics of Fluids*, Prentice Hall Inc., 1991.
51. Prasad, G., Altman M., "Concerns of Adding Mobility Calculations to DIS Exercises," 12th DIS Workshop, Orlando, Florida, March 1995.
52. Preparata, F., "Computational Geometry: An Introduction," Springer-Verlag, 1988.
53. Shabana, Ahmed A., *Dynamics of Multibody Systems*, John Wiley & Sons, Inc., 1989.
54. Shigley, J. E., "Theory of Machines and Mechanisms," McGraw-Hill Pub., 1980.
55. Smith, Carl L., "Commercial Vehicle Performance and Fuel Economy," SAE paper 1970.
56. Stam, J. and Fiume, E., "Turbulent Wind Fields for Gaseous Phenomena", *Computer Graphics proceedings*, pp.369-376, August 1993.
57. Taborek, Jaroslav J., "Mechanics of Vehicles," Machine Design, Part 1-13 May 30 1957, Dec 12 1957.
58. TACOM, "X1100-1c Transmission/AVCR 1360-3c Engine Adaptation program," Report for TACOM by Allison Division of General Motors, June 1982.
59. TACOM, "Steer Duty Cycle Analysis," Report for TACOM by Allison Division of General Motors, March 1985.
60. TACOM, "Investigation of the factors involved in Steering Tracklaying Vehicles," Report for TACOM by Allison Division of General Motors, May 1970.
61. Terzopoulos, D. Platt, J., and Fleischer, K., "Heating and Melting Deformable Models (From Goop to Glop)", *Proceedings Graphics Interface*, pp.219-226, June 1989.
62. Tonnesen, D., "Modeling Liquids and Solids using Thermal Particles", *Proceedings Graphics Interface*, pp.255-262, June 1989.
63. Ts'o, P. Y. and Barsky, B. A., "Modeling and Rendering Waves: Wave-Tracing Using Beta-Splines and Reflective and Refractive Texture Mapping", *ACM Transactions on Graphics*, 6:3,

pp.191-214, July 1987.

64. U.S. Army, "Transmission Assembly and Final Drive Assembly, Model X1100-3B," Technical Manual TM 9-2520-271-34, Oct 1983.

65. Wejchert, J. and Haumann D., "Animation Aerodynamics", *Computer Graphics* 25:4, pp.19-22, July 1991.

66. Wendt, John F., *Computational Fluid Dynamics*, Springer-Verlag New York Inc. 1992.

67. WES, Waterways Experiment Station, Corps of Engineers, Dpt. of the Army, "Cone-Index-Based Estimates of Soil Strength: Theory and User's Guide for Computer Code CIBESS," Waterways Experiment Station", Vicksburg, Tech. Rpt. GL-92-5, May 1992.

68. WES, Waterways Experiment Station, Corps of Engineers, Dept. of the Army, "Mobility Models using Acceleration and Deceleration," Tech. Rpt. GL-88-19, September 1988.

69. WES, Waterways Experiment Station, Corps of Engineers, Dept. of the Army, "Results of Soft-Soil, Acceleration, Turning, And Maneuver Tests With WES M113/2E Hotrod, GM XM1, GM ATR, And A Contemporary M113," GL-83-19, December 1983.

70. WES, Waterways Experiment Station, Corps of Engineers, Dept. of the Army, "The AMC '74 Mobility Model," Tech. Report No. 11921(LL-149), May 1975.

71. WES, Waterways Experiment Station, Corps of Engineers, Dept. of the Army, "The Unified Soil Classification System," Tech. Mem. No. 3-357.

72. WES, Waterways Experiment Station, Corps of Engineers, Dept. of the Army, "Mobility and Plowing Capabilities of the Combat Mobility Vehicle," April 91.

73. WES, Waterways Experiment Station, Corps of Engineers, Dept. of the Army, "NATO Reference Mobility Model Edition II, Users Guide," Tech. Report, June 1992.

74. WES, Waterways Experiment Station, Corps of Engineers, Dept. of the Army, "Steerability Analysis of Tracked Vehicles: Theory and User's Guide for Computer Program TVSTEER," SL-86-30, August 1986.

75. William, J. Plam III, "Modeling, Analysis, and Control of Dynamic Systems," John Wiley & Sons, 1983.

76. Wong, J.Y., "Theory of Ground Vehicles," John Wiley & Sons, 1993.

77. Wong, J.Y., "Terramechanics and Off-Road Vehicles," Elsevier Publications, New York, 1989.

78. Xie, L., P.W. Claar, R.J. Smith, "Computer-Oriented Analytical Dynamics Methodologies for Vehicle Simulation," Proc. 10th Conf. of the ISTVS, 1990.

79. Yong, Raymond N., Ezzat A. Fattah and Nicolas Skiadas, "Vehicle Traction Mechanics," Elsevier Publications, 1984.

0000078

