1-1-1994

Mobility Across Dynamic Terrain - And Engineering Change Proposal To The Dynamic Terrain Testbed Project: Final Report

Mark C. Kilby

INSTITUTE FOR SIMULATION AND TRAINING

N613390-92-K-0001 (P00001)
April 28, 1994

Final Report: Mobility Across Dynamic Terrain - An Engineering Change Proposal to the Dynamic Terrain Testbed Project

Mark Kilby
Lance Marrou
Guru Prasad
Sean Waldron

IST

Institute for Simulation and Training
3280 Progress Drive
Orlando FL 32826

University of Central Florida
Division of Sponsored Research

IST-TR-94-20

INSTITUTE FOR SIMULATION AND TRAINING

Final Report: Mobility Across Dynamic Terrain - An Engineering Change Proposal to the Dynamic Terrain Testbed Project

Contract Number N61339-92-K-1 (P00001)

IST-TR-94-20

Authors:
Mark Kilby
Lance Marrou
Guru Prasad
Sean Waldron

Reviewed by

Dr. Kurt Lin

Art Cortes

Curtis Lisle

Marty Altman

Institute for Simulation and Training • 3280 Progress Drive • Orlando, Florida 32826

University of Central Florida • Division of Sponsored Research

Final Report: Mobility Across Dynamic Terrain

An Engineering Change Proposal

to the

Dynamic Terrain Testbed Project

Contract N61339-92-K-0001 (P00001)

for

U.S. Army Simulation Training and Instrumentation Command (STRICOM)

prepared by

Institute for Simulation and Training
3280 Progress Drive
Orlando, FL 32826

May 2, 1994

Table of Contents

List of Figures

List of Tables

1.0 Introduction

This document summarizes the vehicle mobility and bridge simulation studies conducted by the Institute for Simulation and Training for the U.S. Army Simulation Training and Instrumentation Command (STRICOM). This work, entitled "Mobility Across Dynamic Terrain," was conducted as an extension, or Engineering Change Proposal (ECP), to the project "Dynamic Terrain Testbed Research and Development" under contract N61339-92-K-0001.

1.1 Purpose

This task began as an effort to model certain classes of mobility constraints and mobility aids within the synthetic battlefield. Specifically, IST was tasked with enhancing vehicle simulations with realistic mobility constraints due to terrain and to provide a means of simulating combat utility vehicles, such as the Heavy Assault Bridge (HAB), in tactical scenarios. This work was done in cooperation with the U.S. Army Waterways Experiment Station (USAWES), the U.S. Army Engineer School (USAES), and the U.S. Army Tank-Automotive Command (TACOM). Since the HAB was still under competitive bid at the time of this project, the Armored-Vehicle-Launched Bridge (AVLB), the predecessor to the HAB, was deemed a suitable alternative by TACOM, USAES, STRICOM, and IST.

1.2 Executive Summary

The Battlefield Distributed Simulation-Developmental (BDS-D) Testbed is a networked real-time simulation environment operated and managed by STRICOM for systems acquisition, combat developments and testing. BDS-D builds on the experiences acquired from SIMNET technology development.

A major criticism of the SIMNET system has been its lack of constraints on mobility. In a test at Ft. Knox in 1992, the navigational capabilities of simulated M1A1 and M1A2 vehicles were compared. The usefulness of the test was greatly reduced by the fact that the M1A1 crews could simply drive to the objective, without regard to the intervening terrain except for houses and streams. The new capabilities of the M1A2 were unable to improve on these unrealistically perfect performances, and thus made it difficult to compare the two systems.

As part of a long range effort to improve the realism of networked simulation in BDS-D, IST is presently tasked with building the Dynamic Terrain Testbed, a software system which inserts bulldozers, dynamic soil, running and standing water, minefields and other physically realistic obstacles into the synthetic battlefield. These features have as their principal effects the reduction of mobility in a realistic fashion. However, the original project did not address how these effects would be imposed upon vehicles.

The Mobility ECP was added to this project to address this issue through two important aspects of mobility: bridging and trafficability[1]. These studies have culminated in a software system known as the Tracked Vehicle Simulator, or TVS. This document describes the theory used in developing the vehicle dynamics, mobility, and bridging

models, in addition to the software developed to implement these models. In the course of these studies, significant lessons were learned and areas for future research were identified as described in Section 5.0 "Conclusions".

1.3 Scope of the Document

This document is divided into six sections:

Section 1.0 "Introduction": Describes the purpose of this research as well as providing an executive summary and technical background on the project.

Section 2.0 "Theoretical Model Development": Describes the theory behind the vehicle dynamics, mobility, and bridging models.

Section 3.0 "Software": Provides a high level and detailed description of the Tracked Vehicle Simulation (TVS) software.

Section 4.0 "Data Collection and Testing": Describes the data collection and testing efforts of the TVS for the AVLB.

Section 5.0 "Conclusions": Provides the lessons learned as well as recommendations for future research.

Section 6.0 "References": List of all works cited in this report.

1.4 Project Background

Visual and other behavioral effects, where the simulated earth's shape and properties change in a physically realistic manner are referred to as dynamic terrain. Current commercial image generators do not permit the numerous, arbitrary modifications to the simulated terrain surface required to realistically model the effects of ground-based simulated forces (infantry and armor). Ground warfare involves extensive interaction with, and modification to the terrain. Events such as vehicle tracks, cratering, and construction of berms and anti-tank ditches must be modeled in order to increase training effectiveness. The increased fidelity provided by these types of physical models will be required in future applications of the DIS paradigm such as system test, evaluation, acquisition, doctrine studies, and combined training simulations including constructive, live, and virtual components.

Traditional image generator databases represent the terrain surface with an array of polygons or elevation posts with little consideration for the interaction of vehicles with the soil. IST researchers are working on alternate representations of the terrain surface, real-time physics-based soil and hydrology models, networking protocols, and the development of a testbed to investigate the utility of these new algorithms and data structures within distributed interactive simulations. To provide a more flexible terrain

1. Trafficability is defined as the ability of soil to support the passage of vehicles.

representation that can support a variety of interactions, alternatives to traditional polygonal representations are being explored. The ability of these new representations to support varying resolutions, as well as different environmental properties, across different vendor's image generators and their ability to be integrated into existing and future image generator technology is being evaluated through IST's Dynamic Terrain project.

The Mobility Engineering ECP broadened the scope of the Dynamic Terrain project to study the effects of this dynamic environment on simulated vehicles. Particularly, this task focused on two important aspects of mobility: bridging and trafficability.

Trafficability studies were conducted through the cooperation of the U.S. Army Corps of Engineers Waterways Experiment Station (WES). Since the early 1940's,WES has been studying the problem of vehicle mobility. Through this research, WES has developed a method for determining soil characteristics which is referred to as "cone index". This index, measured by a device referred to as the cone penetrometer, essentially serves as an indication of the strength of the soil to support a moving vehicle. Numerous tests over the years have produced empirical relationships of the cone index with the "go" or "no go" condition of many existing military ground vehicles [Bekker69]. From these relationships, maps have been developed to aid field commanders in determining where they can move in a given area. These mobility predictions are now produced by a software package produced by WES which is referred to as the NATO Reference Mobility Model, version two (NRMMII) [WES92b].

The proposed intention of IST's vehicle mobility research was to adapt the algorithms of NRMMII for use in real-time interactive vehicle simulations. The original NRMMII code was simplified by WES and delivered to IST as a software package referred to as the Simplified Army Mobility Model (SAMM). SAMM predicted the maximum straight-line speed of a vehicle based on the vehicle properties, the type of soil, the strength of the soil (i.e., cone index), and the slope of the terrain in the direction of travel. Many of the simplifications performed to generate SAMM were designed to eliminate those calculations which were typically used to predict driving conditions and strategies that could otherwise be reproduced by other components of an interactive simulation (e.g., obstacle override/avoidance strategies, affects of fog). However, it was discovered early in IST's research that the empirical relationships and algorithms used in SAMM eliminated a need to compute the vehicle dynamics, a necessary component in real-time interactive ground vehicle simulations. Therefore, modifications to the SAMM algorithms were required to incorporate the affects of vehicle behavior that are modeled through the dynamics: acceleration, deceleration, braking, turning, sliding, and transitions due to changing resistive forces (e.g., changing soil types or slopes). These modifications are described in Section 2.6 of this report.

Due to the limitations of the WES model for real-time, a literature search was conducted to locate alternative models for terrain-vehicle interactions. These studies revealed a large body of work from numerous sources which was based on the research of Dr. M.G. Bekker [Bekker60, Bekker69]. Bekker's model provided a more deterministic approach based on the vehicle dynamics and classical soil properties. From this body of work, a second vehicle mobility model was developed which is described in Section 2.7 of this

report.

For both the WES and Bekker mobility models, available terrain data for testing proved difficult to acquire. Requests for cone index databases for Ft. Hunter Liggett, the National Training Center, and other training grounds were made in early 1993. However, data was unavailable for these areas from WES. Furthermore, soil data for the Bekker models are based on classic soil properties which is still difficult to obtain within the field or through laboratory testing according to WES. Therefore, extensive research and measurements would be required to obtain accurate soil property data for these types of models.

However, a temporary solution to the data collection problem is available through use of Interim Terrain Data (ITD). WES has developed algorithms to predict cone indices for an area based on data found in ITD databases. Also, WES has developed methods to estimate classical soil properties used by the Bekker model from cone indices of certain types of soils [WES92a].

As stated previously, a second task of the Mobility ECP was to model bridges and their affect on vehicle mobility. In particular, the U.S. Army Engineer School (USAES) desired a simulation tool to allow them to evaluate the use of new and existing ground-based combat utility vehicles in training scenarios. Through discussions with USAES and the U.S. Army Tank-Automotive Command (TACOM), the new Heavy Assault Bridge (HAB), under development by TACOM, was selected as the vehicle to be modeled. However, it was discovered in the data acquisition phase of this research that a vendor for production of the HAB would not be selected until well near the end of this research.

Since modeling of a competing HAB vendor's design would appear as a premature selection, the Armored-Vehicle-Launched Bridge (AVLB) was agreed upon by STRICOM, IST, TACOM, and USAES as the engineering vehicle to be simulated [USARMY85, USARMY86, USARMY91]. The AVLB is based on an M60 tank chassis and is the predecessor to the M1-based HAB. The AVLB is typically used for rapid crossings of short gaps such as narrow streams, antitank ditches, craters, canals, partially blown bridges or other obstacles. These crossings are accomplished through deployment of a folding bridge which can be retrieved from either end and can span 63 feet. The HAB is designed to extend this spanning capability as well as improve upon other performance criteria.

During the course of the IST Mobility studies, data collection for the AVLB also proved difficult. Requests for AVLB vehicle data were submitted to TACOM which included requests for vehicle mass properties and performance data that would be required in testing the vehicle dynamics model. Only some of the requested data could be provided to IST by TACOM through the original vendor. Additional data was sought through the PM-CATT library with the search providing a technical summaries report which did not provide the needed data on the AVLB[DTIC91]. The lack of available vehicle data lead to the following conclusions: that such data is not required, and therefore, not provided by the manufacturers, or the data is not provided in a central location. Therefore, the required data was estimated based on available data from AVLB technical manuals [USARMY86, USARMY91].

With the available data, models for vehicle mobility and the AVLB were integrated into the Dynamic Terrain Testbed, a flexible software architecture for testing new algorithms and data structures. The architecture is designed so that it can be placed on a network to operate in a Distributed Interactive Simulation (DIS) scenario among multiple participants or can be used with a single simulator. An example of the latest version of this architecture is shown in Figure 1. The architecture consists of special software processors that house the soil, hydrology, and other environmental models and an active database containing the terrain representations. These processors and the database are housed within a Terrain Server/Manager which maintains the terrain state for the DIS simulation scenario. The Terrain Manager can be distributed among one or several simulation nodes on the network with each node consisting of one or many processors. Environmental models are contained within Dynamic Terrain Resources (DTRs). Events such as rain or vehicular motion trigger the appropriate processor (i.e., DTR) that queries and modifies the data representation of the active terrain database, after which, these environmental changes are then transmitted to all simulators participating in the DIS exercise.

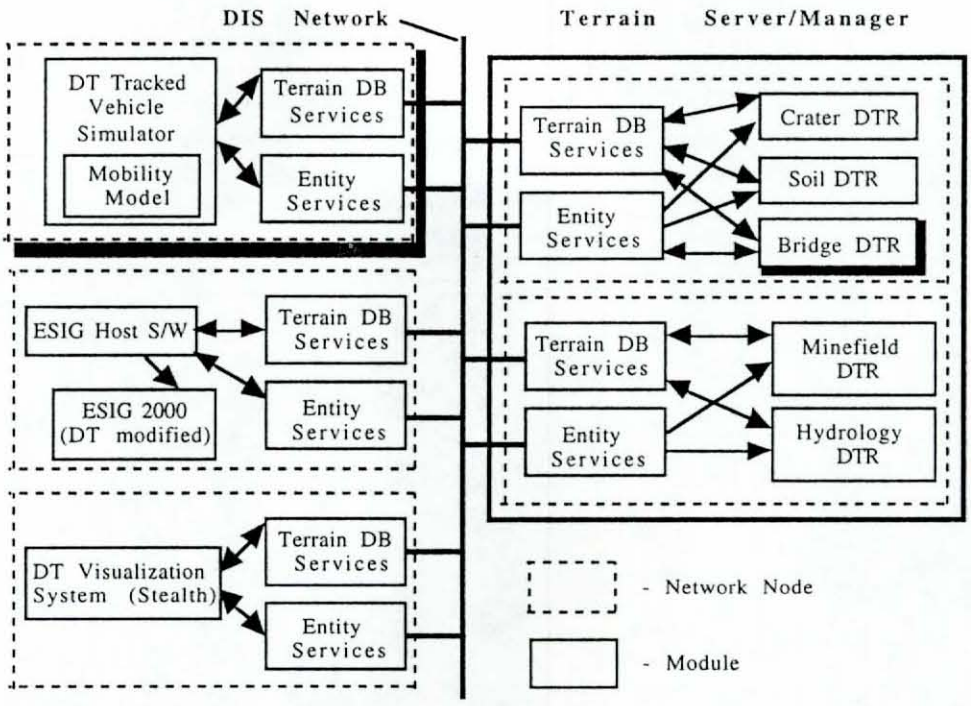


FIGURE 1: Integration of the Mobility and Bridging Models into the Dynamic Terrain Testbed

The environment changing events are monitored through a collection of services [VSL94a, VSL94b]. These services perform the essential bookkeeping of the state of a DIS simulation exercise for each simulation node. Entity Services tracks all entities in the DIS exercise through their entity state PDUs and provides a number of services such as dead reckoning, PDU filtering, and coordinate conversion. Similarly, Terrain Database Services holds the active database and tracks the changes to the state of the terrain. The active database itself stores several attribute layers of terrain information including a

surface attribute layer providing elevation and slope information, a soil type attribute layer, a soil strength attribute layer (be it cone index used by the WES model or classical soil properties used by the Bekker/Wong model), and any other terrain attributes required by the simulation (e.g., moisture content, temperature). Figure 2 illustrates these attribute layers in the database, which are represented by mathematical surfaces. By using mathematical surfaces, the database resolution is independent of the resolution required by the simulator's image generator. These same attribute layers can be transmitted by the Terrain Change PDUs under development through this research.

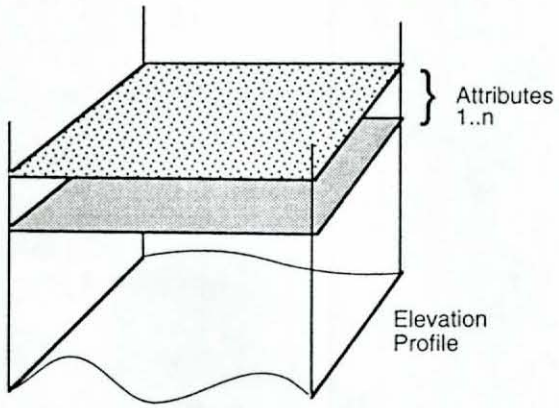
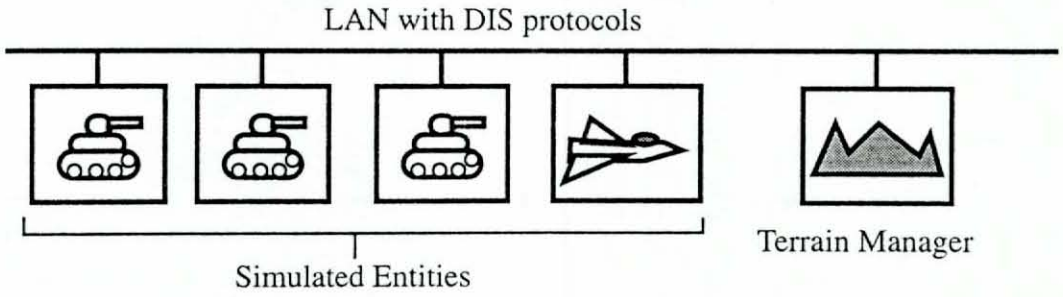
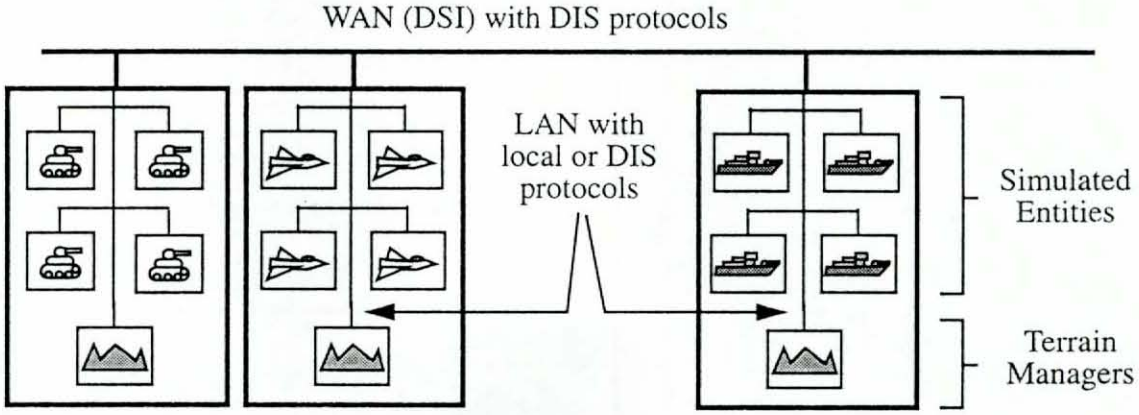


FIGURE 2: Attribute Layers of the Active Terrain Database

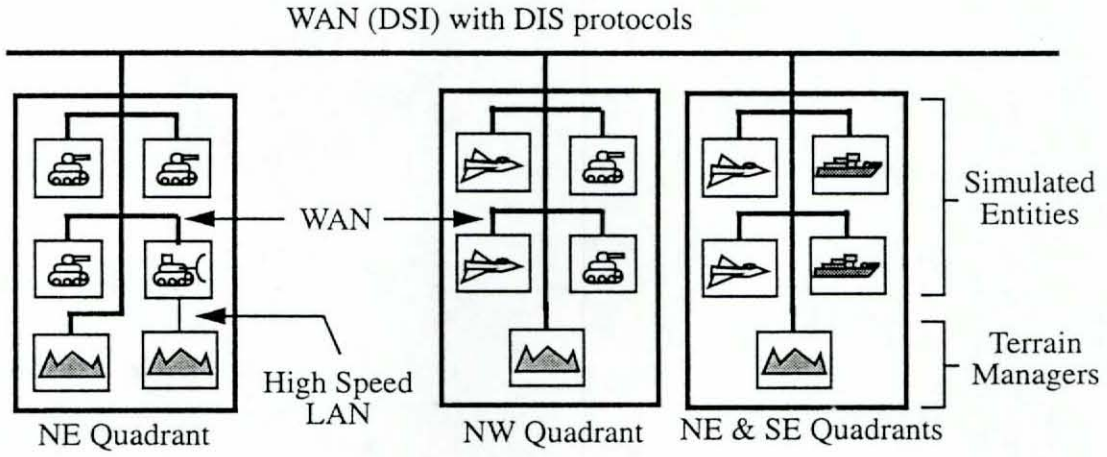
These services can be duplicated on each simulator node to provide a common interface to the dynamic state of the simulation, reduce the complexity of the main simulation applications, and to provide flexibility in the configurations of the DIS simulation exercise. First, it has been recognized by the DIS community that there are common software functions which will be required by each simulator node within a DIS exercise and that these functions can be provided by a common software package [DIS94]. This need is addressed by the services under development within the DT Testbed. Second, by providing this functionality as a separate software package, the complexity of the simulator host software is reduced. Third, having these services on each node provides the flexibility to distribute the functionality of the terrain manager among groups of nodes or each node. This capability reduces network latency and bandwidth requirements in transmitting these environmental changes to all players. Possible configurations of terrain managers within a DIS exercise requiring dynamic terrain are illustrated in Figure 3. Current research within the Dynamic Terrain project indicates that these configurations will be dependent on the exercise scenario. A small collection of simulators may require only a single terrain manager as shown in Figure 3a. For larger exercises, a terrain manager may be provided on each local area network where several simulators may be physically collocated (see Figure 3b). For some scenarios, as shown in Figure 3c, it may prove beneficial to provide terrain managers for separate areas of the virtual gaming area. To determine how these scenarios drive the simulation configuration will require additional research. The development of this architecture, the network protocols, and lessons learned, will be presented in an upcoming paper at the tenth DIS Workshop [Lisle94].



(a) Sample Configuration 1 - Small Collection of Simulated Entities on a LAN with a Single Terrain Manager



(b) Sample Configuration 2 - Groups of Simulators in the Same Physical Location with a Terrain Manager Per Site



(c) Sample Configuration 3 - Groups of Simulated Entities with a Terrain Manager per Virtual Gaming Area and per Simulator with Frequent Terrain Interaction (e.g., Bulldozer)

FIGURE 3: Sample Simulation Configurations for Future DIS Exercises with Dynamic Terrain

Figure 1 also illustrates where the mobility and bridging models are integrated within the testbed. Due to the dependency of the mobility model on the vehicle dynamics of the specific vehicle and the flexibility provided by the Terrain Database Services, the mobility model was directly integrated into the Tracked Vehicle Simulator (TVS). The TVS need only request soil strength, soil type, and other properties required by the mobility and vehicle dynamics models from the Terrain DB Services. The vehicle dynamics also includes models of the AVLB M60 deployment vehicle as well as a model of the bridge deployment mechanism. Dynamics for the bridge (once it is deployed), is handled by the Bridge DTR within the Terrain Manager. Once the bridge is deployed, it acts as a separate entity with the bridge dynamics and the bridge entity state being managed by the Bridge DTR. This allows the bridge to react to multiple vehicles driving over it, collapsing under excessive loads or through detonations. The theoretical models for the vehicle dynamics, bridge dynamics, and the mobility models are described in Section 2.0. The software which implements these models in the Tracked Vehicle Simulator and the Bridge DTR is described in Section 3.0.

2.0 Theoretical Model Development

2.1 Vehicle Kinematics

The initial stage in the design of any vehicle simulator is the development of a format for representing the vehicle's geometry and prescribing it's motion. This format, known as the kinematic representation of the vehicle, provides the basis for all of the algorithms used to manipulate the vehicle during a simulation. The kinematic representation also provides a data standard for communication between different parts of the simulation system and also for sending data across networks.

2.1.1 Theoretical Foundations

Kinematic modeling and analysis is concerned with the motion of a rigid bodies without any regard to forces and moments. In other words, kinematics involves describing all the possible ways that an object can move (i.e., it's degrees of freedom) and how that motion is related to other coordinate systems, especially databases and articulated parts.

2.1.2 Derivations for Real-time Performance

As shown in Figure 4(a), the axial convention used for the Tracked Vehicle Simulator places the positive x-axis through the vehicle's "starboard" side, the positive y-axis out the front of the vehicle, and the positive z-axis up through the top of the vehicle. The origin of the vehicle's body coordinate system is placed on the bottom of the vehicle, symmetrically front-to-back and side-to-side with reference to the vehicle's hull. This is the standard form used for vehicle models based in Performer and GL (the image generation software that TVS utilizes).

To describe the vehicle's orientation, two different methods are used. Performer and GL utilize one of the Euler Angle sequences as shown in Figure 4(a). This is the same sequence as used by SIMNET and many other simulation systems. The dynamics equations, however, require a more robust formulation that avoids mathematically undefined orientations that are physically realizable. Such orientations are referred to as singularities. Therefore, the dynamics equations utilize quaternions, or Euler Parameters, to specify the vehicle's orientation [Minkler, NASA76]. A quaternion is a mathematical construction for describing an arbitrary angle of rotation about an arbitrary normal vector in space as shown in Figure 4(b). The internal rotation matrix of the vehicle used by the vehicle dynamics simulation is determined by quaternion manipulations. When the dynamics simulator interfaces with the rest of the Tracked Vehicle Simulator, it decomposes (with standard matrix manipulations) the internal rotation matrix so as to produce an equivalent set of euler angles in the format of the graphical system.

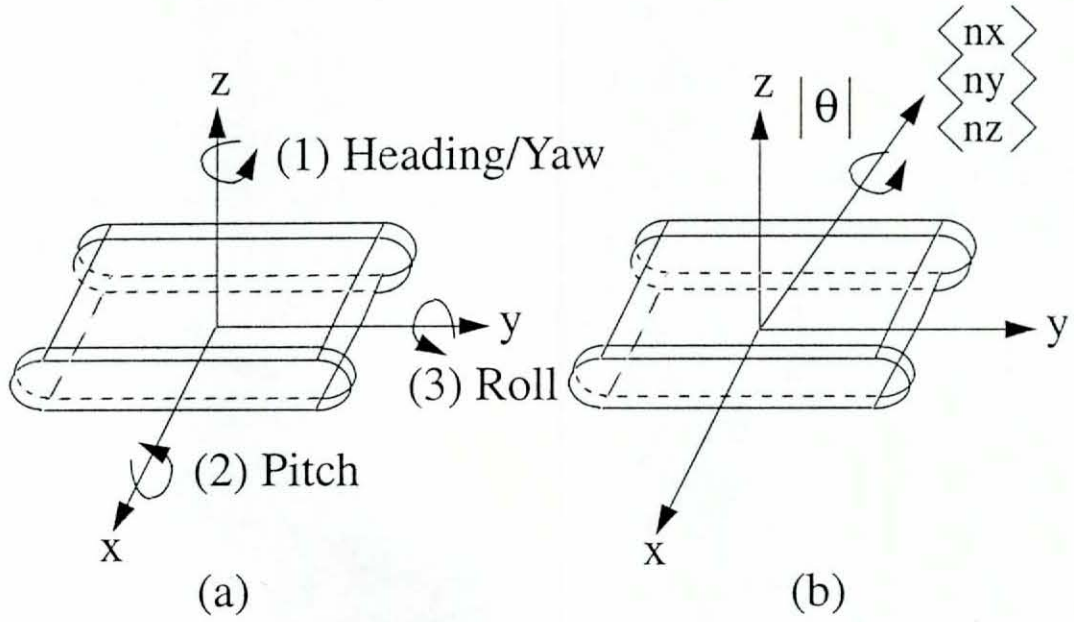


FIGURE 4: Coordinate Axes for the Tracked Vehicle Simulator

2.1.3 Assumptions and Limitations of the Model

The quaternion formulation is more mathematically based than the euler angles, so using quaternions actually speeds up the simulation process. This benefit is offset due to the time required to translate the results into euler angles for output. However, the gain in accuracy and stability (unlike euler angles, quaternions have no singularities) is more than enough reason to justify using quaternions. The dynamics model and graphical model both use meters for length measurements and thus are identical, producing no additional processor overhead.

2.2 Vehicle Dynamics

The tracked vehicle simulator is based on the dynamically correct equations of motion for a generic, rigid body with six degrees of freedom. Continuous, numerical solution of these equations of motion allows the vehicle's position and orientation to be regularly updated in a realistic manner. In addition, the velocity and acceleration of each of the degrees of freedom is also available with minimal extra computational effort.

2.2.1 Theoretical Foundations

The primary purpose of the vehicle dynamics model is to provide as realistic as possible of a simulation of the actual vehicle's behavior. The Tracked Vehicle Simulator does this by modeling the vehicle with the Newton-Euler equations of motion. This allows for accurate simulation based on the mechanical properties of the vehicle. The propulsion and support of the vehicle are determined by applying external forces and moments onto the vehicle's

hull through the tracks and the suspension components. Additional forces and moments may also be applied to the vehicle by simply adding them to those external forces and moments already used. Therefore, no change in the solution methodology is required. Dynamically correct modeling also provides a basis for dynamically correct component articulation.

The origin of the database acts as an inertial reference frame for the vehicle, thus allowing the dynamics equations to be developed and solved using the standard Newton-Euler approach. The vehicle's body is then used as a non-inertial, but established, reference frame for the articulation equations.

2.2.2 Derivations for Real-Time Performance

The real-time performance criteria of a simulator requires that the equations of motion be optimized for high speed and efficiency. To do this, a special characteristic of bridging vehicles is utilized; no part articulation is allowed during driving. Since bridging vehicles only articulate the bridging systems while parked for deployment (e.g. the AVLB) this is reasonable. This assumption allows the reference point for the dynamics equations to be the vehicle's center of mass. An offset vector is built into the equations that allows the equations to be resolved around the center of mass while still reporting data in terms of graphical coordinate system at the base of the vehicle. This decouples the translational equations of motion from the rotational equations of motion and also from each other. The rotational equations of motion are further simplified by pre-inverting the vehicle's inertia matrix prior to simulation. Since the mass, center of mass, and rotational inertias are drastically different when the bridge is mounted and when it is deployed, two cases are developed for the dynamic properties of the vehicle. This is illustrated in the following equations.

$$a_X = \frac{f_X}{m} \quad \text{(EQ 1)}$$

$$a_Y = \frac{f_Y}{m} \quad \text{(EQ 2)}$$

$$a_Z = \frac{f_Z}{m} \quad \text{(EQ 3)}$$

$$\alpha_X = M_X C_{XX} + M_Y C_{XY} + M_Z C_{XZ} \quad \text{(EQ 4)}$$

$$\alpha_Y = M_X C_{YX} + M_Y C_{YY} + M_Z C_{YZ} \quad \text{(EQ 5)}$$

$$\alpha_Z = M_X C_{ZX} + M_Y C_{ZY} + M_Z C_{ZZ} \quad \text{(EQ 6)}$$

given,

$$\begin{bmatrix} C_{XX} & C_{XY} & C_{XZ} \\ C_{YX} & C_{YY} & C_{YZ} \\ C_{ZX} & C_{ZY} & C_{ZZ} \end{bmatrix} = \begin{bmatrix} I_{XX} & I_{XY} & I_{XZ} \\ I_{YX} & I_{YY} & I_{YZ} \\ I_{ZX} & I_{ZY} & I_{ZZ} \end{bmatrix}^{-1} \quad \text{(EQ 7)}$$

where,

a_X is the sideways translational acceleration (in meters per second)

a_Y is the forward translational acceleration (in meters per second)

a_Z is the vertical translational acceleration (in meters per second)

F_X, F_Y, and F_Z are the net applied forces (in newtons)

m is the mass of the vehicle (in kilograms)

α_X is the pitching acceleration (in radians per second)

α_Y is the rolling acceleration (in radians per second)

α_Z is the heading acceleration (in radians per second)

M_X, M_Y, and M_Z are the net applied moments (in newton-meters)

I_{XX}, I_{YY}, and I_{ZZ} are the principal moments of inertia (in kilogram meters squared)

I_{XY}, I_{YZ}, I_{ZX}, I_{XZ}, I_{ZY}, and I_{YX} are the cross-products of inertia (in kgm^2)

The C_{NN}'s are the resultant coefficients from inverting the inertial matrix (in kilogram meters squared inverted, $1 / kg\ m^2$)

One obstacle to developing a real-time simulator is the magnitude of the time step that can be used per iteration of the dynamics calculations. The implementation of this project requires that the vehicle dynamics simulation must be able to operate on the same processor as the image generator. This can limit the simulation time step to as much as a fifth of a second on our developmental machines. Even using a faster processor, the time step may still be longer than straightforward methods can adequately handle.

To simulate motion, a vehicle's position, orientation, velocities, and accelerations must be predicted across each time step by numerically integrating it's equations of motion. The vehicle dynamics simulator uses a modified version of Euler's Method for numerically integrating the equations of motion [Nakamura91, Press91]. The Modified Euler's Method functions as follows. First, the state of the vehicle at the current time step is evaluated and all the net forces and moments acting on the vehicle are determined. Next, the current acceleration of each degree of freedom is calculated by solving the dynamics equations listed above. Then the vehicle's state at the end of the current state is determined by assuming that these accelerations remain constant. Finally, this new state is reevaluated to determine what the accelerations should be at this state. The difference between the original accelerations and the estimated ones are used to improve the estimation of the acceleration during the time step. The process is illustrated below.

- test velocity = old velocity + (old acceleration * time step)
- test DoF = old DoF + (old velocity * time step) + (old acceleration * time step squared)
- test acceleration = "acceleration evaluated at new velocity and DoF"
- new acceleration = weighted average of old acceleration and test acceleration
- new velocity = old velocity + (new acceleration * time step)
- new DoF = new DoF + (new velocity * time step) + (new acceleration * time step squared)

This method has three major advantages. First, it is computationally compact and does not require idealistic information. Second, it has inherent numerical stability, unlike the standard Euler's Method with simple integration. Third, the trial state derived for the initial acceleration estimation provides a means of sampling the environment at the next position and orientation. Thus, the effects of location based changes in the environment may be combined with the effects due to dynamic changes in the vehicle's state to give a more accurate indication of the next state.

2.2.3 Assumptions and Limitations of the Model

The Tracked Vehicle Simulator has a few key limitations due to some basic assumptions and the project specifications. First, the vehicle is modeled as a rigid body; meaning that body cannot undergo any deflections. While this is acceptable for normal driving, it means that no effects on the vehicle due to extreme loadings, such as high speed collisions or drops from significant heights, can be modeled. There is also no way to correctly model the effects of explosions or detonations on the vehicle. However, this is the most common and acceptable approach in real-time vehicle modeling; deflection modeling comes under the scope of structural analysis.

Second, the vehicle can only interface with terrain and bridges. This interface is made only through contact with the vehicle's tracks. This is all that is necessary in order to drive the vehicle. Other types of mechanical interactions are possible, but they have not yet been added to the vehicle dynamics simulator.

Third, the large time step of the simulation system often causes a loss in fidelity when modeling short duration events such as small object bumps of quick changes in terrain geometry. While small bumps and the like are not significant, sharp terrain changes may make predicting motion difficult in real-time. When an impossibility is detected, such as penetrating a hill or wall, a set of system safeties activates. These are routines that kinematically place the vehicle over the terrain. While this does allow some loss of dynamic correctness, it does provide continuity of simulation. It is difficult to model all interactions in real-time because of the limitations of processor speed.

A final limitation of the model is that articulations are not currently possible while driving. The simplifications made to the vehicle dynamics equations, to allow faster development of the tractive effort algorithms, prevent on-the-fly articulations. As this is not needed for bridging vehicles, this limitation was not a problem during development. However, only modest effort would be required to add such abilities in the future due to the modularity of

the software design.

2.3 Simulation of Internal Systems

While the dynamics and mobility algorithms are sufficient for describing the behavior of the vehicle's hull (i.e., it's gross body motion), there still needs to be an interface to the operator for control of the vehicle. Since internal components of track vehicles vary a great deal from vehicle to vehicle, modeling of the internal systems is performed separately from the dynamics and mobility equations which are flexible and reusable.

2.3.1 Theoretical Foundations

Two connections are used to provide the fundamental interface between the vehicle's internal systems and the physical world of the simulator: the left and right drive sprockets. The speed of the drive sprockets should be jointly determined by the mobility equations and the internal systems of the vehicle, namely the engine, torque convertor, transmission, differentials, steering, and brakes. In order for any simulator to accurately model a tracked vehicle's performance, the response of these systems is necessary. Otherwise, the user's control inputs will not have a consistent relationship to the response of the controls on the actual vehicle.

Unfortunately, vehicle systems are not generic in nature, many different modeling relationships are required to model all of the possible configurations commonly used by the manufacturers of most vehicles. For this project, a simplified internal systems model was constructed using the AVLB's basic specifications as a model [USARMY86, USARMY91]. Although many of the algorithms can be used to model other vehicles, the organization and combinations used in the Tracked Vehicle Simulator only correspond to the AVLB. Since the purpose of this project is the development of mobility modeling, not a generic vehicle system simulation, this was considered an acceptable trade-off.

2.3.2 Derivations for Real-time Performance

In order to run this system in real-time, several simplifications were made to the internal systems so as to give the mobility and dynamics algorithms as much processor time as possible. Due to our lack of specific data on the AVLB's systems, no truly detailed model was possible. Therefore a kinematic, not dynamic model of the vehicle's systems was constructed. This model uses the following procedure. First, the throttle input is read in from the user interface of the TVS. Then, the engine responds to any difference between it's current rpm and the rpm naturally corresponding to the throttle position. Next, the torque convertor similarly adjusts to any differences between it's rpm and that of the engine driven flywheel. Transmission effects next come into play, including the current gear and/or the current state of the gear changing actuators. Then comes the effects of the main differentials and shafts. The position of the steering handlebar is then read in to determine the state of the steering differentials. After this, rotation is transferred to the drive sprockets, where braking effects and feedback from the mobility equations are considered. The final result is the wrapping speed of each track, the theoretical speed of

the vehicle for each track. This is the main input to the tractive effort algorithms. The process is detailed below.

The first stage in the vehicle controls system, the user chooses the throttle position. This is passed into the vehicle dynamics manager, which also functions as the internal systems simulator. The simulator compares the throttle opening to the current state of a kinematic representation of an engine. If the speed of the engine does not correspond to the throttle position, the speed of the engine changes by some rpm step size, a simple kinematic constant. The torque convertor then compares it's speed to that of the engine and similarly makes necessary changes. Then the transmission follows suit. Changing gears controlled by the user, with the exception some shift locks. These shift locks are set when the transmission detects to great of a speed disparity between what speed it needs to turn and how fast its input or output shafts are turning. After the transmission, the differentials and final drive ratio are modeled similarly to the torque convertor. At this stage the steering ratio is also considered, just as the left and right drive speeds are being separated. The steering ratio is controlled by the user. It is used to bias the drive speed to one side or the other. The final step is brake simulation. This is done independently for each drive sprocket, even though the same brake value is read and user for each side. This provides the drive sprocket speeds for use in the mobility equations. All of the kinematic step factors used in the internal systems simulation were matched to all the available AVLB performance specifications obtained from the limited reference data that we had on its systems. The current model is as accurate as it can be without more detailed performance specifications. Variations between the model and actual performance also cannot be calculated until better data is available.

Originally, the internal systems were intended to be a dynamic simulation based on real performance data. This would have allowed the drive torque applied at each drive sprocket to be modeled. Thus, the speed of the drive sprocket could be determined with dynamic equations of motion, for added realism. Unfortunately, the lack of data prevented a true model from being designed. Therefore the kinematic model was developed.

2.3.3 Assumptions and Limitations of the Model

As described earlier, the internal systems model is a kinematic representation of the AVLB's systems as developed from the AVLB's operator's and technical manuals [USARMY86, USARMY91]. While it suffices for running the vehicle, a more descriptive model of the vehicle's equipment must be developed before a more accurate simulation is possible. The dynamics and mobility algorithms respond correctly to whatever input is provided by the vehicle systems model. However, the vehicle systems model used in this case may not represent the actual performance of the AVLB due to the lack of available vehicle specifications.

2.4 Bridge Articulation Dynamics

The next phase in the vehicle simulation is the modeling of the launcher's articulations. While the dynamic modeling of the vehicle is generic and reusable, the bridging system is, by necessity, specific to the AVLB application. However, the integration standards that

were developed for the AVLB have enabled other vehicles with articulating parts to be modeled with minimum redesign. In addition, the modeling of the bridge components used for the articulation analysis was vital in modeling the behavior of the deployed bridge in the Dynamic Terrain Testbed. The simulation benefits of proper dynamic modeling still hold true for articulating large components.

2.4.1 Theoretical Foundations

The dynamics of the articulated components of the AVLB are driven by equations similar to those employed in the vehicle dynamics code. The components of the bridging system are treated as rigid bodies that rotate in the vehicle's body coordinate system. The bridge components are stored with local coordinates and quaternions. Refer to the figures in the following section. The global coordinates and orientation of the articulated components are available with limited conversions, thus allowing direct communication with the Tracked Vehicle Simulator's environmental database manager (ENV_Mgr). Because the dynamic state and configuration of the vehicle are available through the object-oriented program structure, the bridge articulation dynamics can interface with the vehicle hull in a dynamically correct manner. By using the vehicle as a step-reference frame, the bridging system can use the database origin for an inertial reference frame.

2.4.2 Derivations for Real-time Performance

As in the vehicle dynamics and mobility algorithms, the algorithms used for simulating the bridging systems has been optimized to take advantage of the specific configuration of the AVLB. This resulted in three separate dynamics algorithms for modeling the AVLB. All three share the same dynamics and geometric data, while each takes advantage of a special case to reduce the number of calculations per simulation time step. Thus many operations unnecessary for each bridging operation are removed, leaving only the minimal number of equations required. These simplifications significantly advanced development of the methods and protocols required to simulate the AVLB's bridging systems. See the figures below.

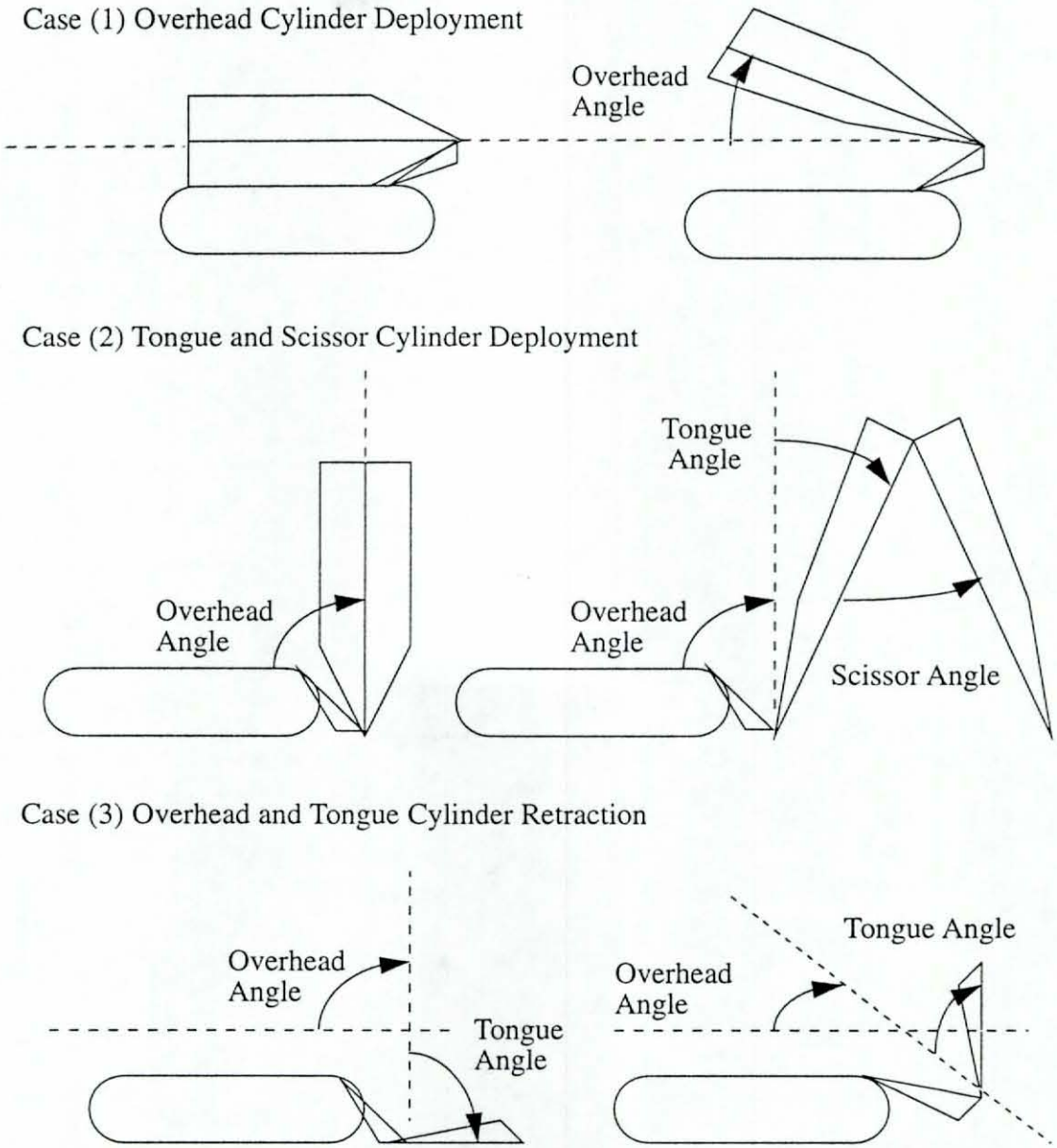


FIGURE 5: Three Stages of AVLB Bridge Deployment Systems

The three stages of AVLB bridge deployment on which these algorithms are based are shown in Figure 5. The overhead deployment algorithm controls the deployment of the entire, mounted bridge assembly off of the top of the vehicle and into an upright position. This method also retracts the bridge back on top of the AVLB. The tongue and scissor algorithm deploys the bridge from this upright position in an attempt to position the bridge on the ground. The tongue and overhead method models the retracting cylinders after the bridge has been detached. All three methods use ground contact and interface methods derived from the mobility model for use in modeling the restrictive effects of terrain

contact with the deploying bridge and the outrigger landing used for support. The bridge articulation simulator contains extra algorithms for use in detaching the bridge, and spawning a networked bridge counterpart, and reattaching it, and removing the networked bridge. These methods uses kinematic data from the bridging systems, which do not need to use dynamics. They are important for data management and are explained in the software design for the Vehicle Dynamics Manager. The basic dynamics equations used in these three cases are the angular form of Newton's Second Law, i.e., the sum of the moments equals the moment of inertia multiplied by the angular acceleration. The solution and prediction methods are the same ones used in the vehicle dynamics equations.

2.4.3 Assumptions and Limitations of the Model

The primary assumption of the bridging model is that only the AVLB will be modeled by this system. Modeling another bridging system, or any other articulation system, will require some modification to the articulation code. However the basic dynamics and interfaces to the vehicle hull and to the environment interfaces are standardized. Another vehicle was modeled by the Tracked Vehicle Simulator and it was able to function using the same interfaces, but with different articulation routines. A different hydraulically driven bridging system could probably be adapted with only modest effort.

The main limitation of the model is its restriction to the power and ability of the vehicle dynamics. Since the current version of the vehicle dynamics code does not support specialized articulations, the mobility model cannot properly run during bridge deployment. That is, the vehicle dynamics model assumes no change in the configuration (i.e., the inertial properties) of the vehicle while driving. This restriction allows for a generic vehicle dynamics and mobility model. However, the vehicle model could easily be modified for different vehicle configurations. This was done to allow for direct reusability of the vehicle model in the immediate future since most other simulated vehicles will experience minimal configuration (i.e. inertial) changes. Until more complete and flexibly generic dynamics code is available for dynamic terrain simulations, these methods will provide a basis for vehicle modeling.

2.5 Deployed Bridge Dynamics

Although it is not directly a part of the Tracked Vehicle Simulator (TVS), the deployed bridge simulator, or Bridge DTR, and all of it's dynamics were developed using the same methods as the TVS. Likewise, all of the interfaces between the vehicle and the bridge were developed concurrently.

2.5.1 Theoretical Foundations

The deployed bridge was modeled with a modified version of the Tracked Vehicle Simulator's dynamics and mobility algorithms. The same dynamics and tractive effort equations were used; however, simplifications were made based on the nature of the bridge's particular structure. When deployed on the terrain, the bridge basically acts the same as a tracked vehicle with a different mechanical configuration and structure; almost like a long, narrow vehicle with long tracks that don't move relative to the body. The

constants that describe the rigid body's mass, center of mass, moments of inertia, structural geometry, and contact surfaces were simply changes to reflect the specific nature of the particular bridge's structure.

2.5.2 Derivations for Real-time Performance

The only significant difference between the deployed bridge simulator and the vehicle dynamics was that they are controlled by different processes. The deployed bridge was not run as part of the Tracked Vehicle Simulator. Instead it was run separately by a part of the Dynamic Terrain Testbed. The deployed bridge was run by a bridge manager, or Bridge DTR, that handles multiple bridges simultaneously. This system provided each simulated bridge with all necessary data for the environment and other vehicles.

The Bridge DTR used entity data to scan for other entities that may be in contact with the bridge. By tracking the position and velocity of each of these entities, a rough estimation of the forces transmitted between each entity and the bridge was possible. This was accomplished by using the vehicle's acceleration in a basic dynamics equation. Assuming the mass of the vehicle can be found, or calculated from weight, the force applied by passing vehicles was estimated simply through Newton's Second Law, i.e.,

Force applied = (mass of the vehicle) * (acceleration of the vehicle) (EQ 8)

This was a crude method, but it was the best that is possible with the data which could be transferred across a DIS network. At each time step, data on the vehicles acceleration may be given. If not, it was estimated from changes in velocity, or worst case, from changes in position compared over time. When the Bridge DTR activated each deployed bridge, they each scan the entities, estimate applied forces, and then react in a dynamically correct manner. Because the Bridge DTR interfaced with the network in a different manner, some of the unit conversions were performed differently. Otherwise, there were no major theoretical differences in functionality between the Bridge DTR, and the vehicle dynamics of the Tracked Vehicle Simulator.

Modeling failure criteria was much easier. Assuming that the vehicle's weight was available, that weight was applied to the bridge whenever a vehicle was located on top of it. A simple test then compared that weight to some preset maximum load. Exceeding this load caused the bridge to collapse and be destroyed.

2.5.3 Assumptions and Limitations of the Model

Most of the deployed bridge's assumptions and limitations were the same as those made for the Tracked Vehicle Simulator. The only difference was in the use of interaction algorithms that estimate how other vehicles were mechanically affecting the bridge. These proximity and contact estimation mechanisms were simple and could have caused problems with complicated scenarios, such as vehicles pushing on the side of the bridge or multiple entities trying to affect the bridge at the same time. Unfortunately, the current DIS paradigm limits the amount of interaction between a bridge and the vehicles near it due to the restrictions on the amount of dynamic data that cn be transferred across a DIS network. Without direct information sharing, accurate dynamic simulation is not yet

possible because the actual forces and moments in the interactions cannot be communicated across the network. Dynamically correct equations of motion cannot function accurately if the external forces and moments supplied to them are not entirely accurate.

2.6 WES-derived Real-time Mobility Model

A major goal of this project was to provide realistic mobility constraints on simulated ground vehicles. The planned approach was to modify the Army's existing mobility model, the NATO Reference Mobility Model (NRMM II), for real-time performance and to incorporate the revised model into a real-time vehicle simulation. This model has been developed by the U.S. Army Corps of Engineers' Waterways Experiment Station (WES) [WES92b].

Before explaining the modifications made to NRMM II, it is important to understand the model itself. This model is based on empirical relationships between measurable terrain properties and the soil-independent (i.e., theoretical) vehicle power output. These relationships are typically represented by the curves shown in Figure 6 [WES92b]. As shown in the figure, these empirical relationships are based upon associations relating the tractive force at the wheel to the speed at the wheel rim. This relationship is represented by two curves. The first, Curve A, represents average tractive force versus speed based solely on the theoretical power output of the vehicle. This theoretical estimate is based on the engine type, the transmission, power train, and internal frictional losses. The second curve, Curve B, represents the effects of soil resistance and terrain slope on Curve A. After these calculations, other limiting factors are considered. For instance, limiting speeds due to visibility, obstacle avoidance or override strategies, and surface roughness produce maximum speed limits on the speed axis. On the force axis, limitations due to vegetation and other resistive forces are considered. From these two groups of limiting factors and Curve B, the maximum predicted speed of the vehicle through a homogenous area of terrain is estimated. It is important to note that this prediction is for a maximum speed only and does not indicate the range of speeds the vehicle will achieve through normal operation (e.g., steering, acceleration, deceleration and braking). Thus, vehicle dynamics are not considered within this model.

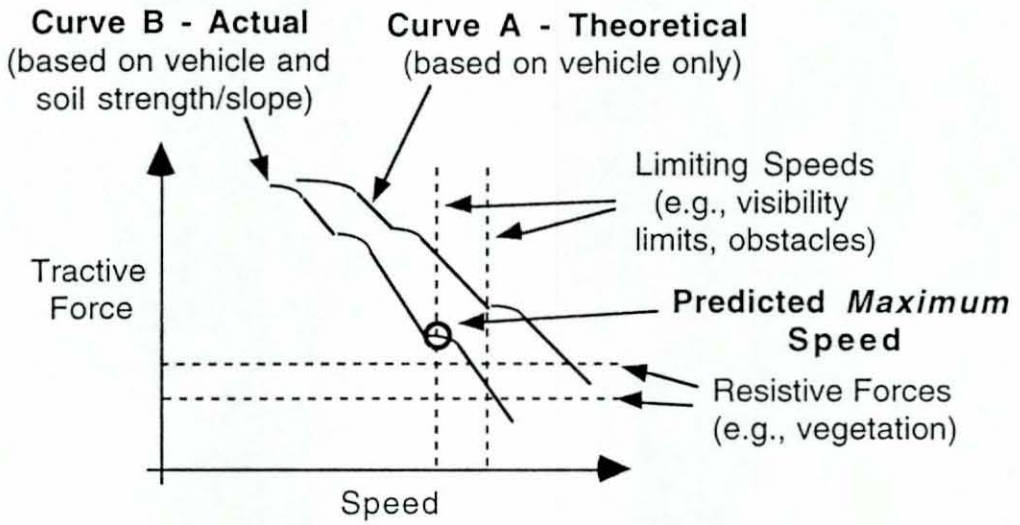


FIGURE 6: Example of Tractive Force versus Speed Curve used by the NATO Reference Mobility Model

The following sections explain how the empirically developed NRMM II model has been modified for real-time performance.

2.6.1 Assumptions

The terrain conditions are evaluated using cone index which is measured by a cone penetrometer and is used in predicting the vehicle performance. One of the parameters developed at U.S Army Corps of Engineers Waterways Experiment Station (WES) to evaluate the vehicle mobility is called the vehicle cone index (VCI). VCI is used to predict the vehicle performance on various types of soils. VCI can be defined as the minimum amount of soil strength, in the upper critical layers, for a vehicle to successfully make a specific number of passes. The following definitions are used in deriving the equations of motion for the mobility model. The mobility model discussed in this section, which uses cone index (CI) to obtain WES derived soil properties, will be henceforth referred as the trafficability model.

2.6.1.1 Mobility Index

Mobility Index (MI), a parameter depending on the vehicle features is needed to evaluate the VCI. The MI of a tracked vehicle is defined as

$$MI = \frac{(F_{Contact\ Pressure} \times F_{Weight} + F_{Bogie} - F_{Clearance}) \times F_{Engine} \times F_{Transmission}}{F_{Track} \times F_{Grouser}} \quad \text{(EQ 9)}$$

Where the Contact Pressure Factor,

$$F_{Contact\ Pressure} = \frac{\text{Gross Vehicle Weight (lbs.)}}{\text{Area of Tracks in contact with Soil}(in^2)} \quad \text{(EQ 10)}$$

and the Weight Factor is

$$\begin{aligned} F_{Weight} &= 1.0 \text{ (vehicle weight} < 50{,}000 \text{ lbs.)} \\ &= 1.2 \text{ (50,000 lbs.} \le \text{vehicle weight} \le 69{,}999 \text{ lbs.)} \\ &= 1.4 \text{ (70,000 lbs.} \le \text{vehicle weight} \le 99{,}999 \text{ lbs.)} \\ &= 1.8 \text{ (vehicle weight} \ge 100{,}000 \text{ lbs.)} \end{aligned} \quad \text{(EQ 11)}$$

and the Grouser Factor is

$$\begin{aligned} F_{grouser} &= 1.0 \text{ (Grouser} < 1.5\text{“)} \\ &= 1.1 \text{ (Grouser} > 1.5\text{“)} \end{aligned} \quad \text{(EQ 12)}$$

and the Track Factor is

$$F_{Track} = \frac{\text{Track width ,in}}{100} \quad \text{(EQ 13)}$$

and the Bogie Factor is

$$F_{Bogie} = \frac{\text{Gross weight ,lb/10}}{\text{Total No of Bogies on Tracks x Area, in}^2 \text{ of 1 track shoe in contact with ground}} \quad \text{(EQ 14)}$$

and the Transmission Factor is

$$\begin{aligned} F_{Transmission} &= 1.0 \text{ (Automatics)} \\ &= 1.05 \text{ (Manual)} \end{aligned} \quad \text{(EQ 15)}$$

and the Clearance Factor is

$$F_{Clearance} = \frac{\text{Clearance ,inches}}{10} \quad \text{(EQ 16)}$$

and the Engine Factor is

$$\begin{aligned} F_{Engine} &= 1.0 \text{ (Hp/ton of Veh. Wt.)} \\ &= 1.05 \text{ (Hp/ton of Veh. Wt.)} \end{aligned} \quad \text{(EQ 17)}$$

2.6.1.2 Vehicle Cone Index

As stated previously, VCI is the minimum soil strength required to support the movement of the vehicle. The value of VCI, developed at WES, is defined by the following empirical relations:

For Muskeg soil,

$$VCI_{mk} = 13 + 0.25 * \text{Contact Pressure} \quad \text{(EQ 18)}$$

For 50 passes,

$$VCI50_{mk} = 14 + 3.4 * \text{Contact Pressure} \quad \text{(EQ 19)}$$

For Fine grained soil,

$$VCI_{fg} = 7.0 + 0.2 * MI - \frac{39.2}{MI + 5.6} \quad \text{(EQ 20)}$$

and for 50 passes on fine grained soil

$$VCI50_{fg} = 19.27 + 0.43\ MI - \frac{124.79}{MI + 7.08} \quad \text{(EQ 21)}$$

2.6.1.3 Cone Index

Cone Index (CI) is the measure of soil strength obtained by field measurements using the cone penetrometer device developed at WES (See Figure 7). CI is used in formulating the empirical relations governing the vehicle motion. It was one of the quickest and simplest methods to determine the vehicle mobility (i.e., GO/NOGO condition) [McRay64]. Rating cone index (RCI) is defined as the product of CI and the remolding index. The remolding index provides a measure of the soil compaction occurring with many vehicle passes. CI gives a comprehensive measure of the shear and normal stresses on the cone face of the penetrometer (i.e., cohesion, shear modulus, angle of internal shearing resistance, etc.).

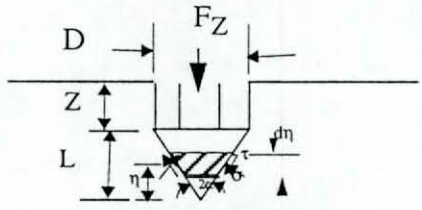


FIGURE 7: Geometry and Stresses on a Frustrum of a Cone

Using static equilibrium, the resistive force, Fz, acting on the cone is given by,

$$F_Z = \int_0^L (\sigma \tan\alpha + \tau)\, 2\pi r\, d\eta \quad \text{(EQ 22)}$$

where σ is the normal stress, α is the cone angle, τ is the shear stress and $r = \eta \tan \alpha$

The Cone Index could then be defined as

$$CI = \frac{F_Z}{\pi R^2} \quad \text{(EQ 23)}$$

Excessive soil strength is the difference of CI and VCI, and is a parameter used in obtaining the towing resistances from the empirical relations of WES. CI values and the depth of penetration, z, are required for the terrain on which the vehicle mobility is to be predicted.

When the cone index is not available in the database, it can be computed from the other available soil properties. For instance, properties used by a Bekker mobility model can be converted to a cone index if an average penetration depth of six inches is assumed [Bekker69]. The more accurate representation of CI is computed using expressions developed by WES [WES88a] and is given by,.

$$CI = 6\tilde{G}^m \frac{\tan\alpha}{\tan\phi}\left(\frac{1+\sin\varphi}{3-\sin\phi}\right)\left[\frac{\tan(\alpha+\tan\phi)}{(R\gamma\tan\phi)^2}\right]\Omega - C\cot\phi \quad \text{(EQ 24)}$$

$$\text{where } \Omega = \frac{[(Z+L)\gamma\tan\phi]^{3-m} - [C+(z+L)\gamma\tan\phi + (2-m)L\gamma\tan\phi](C+z\gamma\tan\phi)^{2-m}}{(2-m)(3-m)} \quad \text{(EQ 25)}$$

$$m = \frac{4\sin\phi}{3(1+\sin\phi)} \quad \text{(EQ 26)}$$

and

α = Cone Angle,

C = Cohesion,

z = Depth of penetration of cone,

ϕ = Apparent angle of internal friction,

γ = Soil Density,

$\tilde{G}$ = Apparent rigidity modulus

$$\tilde{G} = 0.5\left[A + \frac{1-Be^{-\beta z}}{1+Be^{-\beta z}}\right]G \quad \text{(EQ 27)}$$

A,B, β = Cone constants

A = 0.986

$B = 100$

$\beta = 0.55$ in.$^{-1}$

G = Rigidity Modulus

The rigidity modulus, G, at ambient pressure level,($\sigma = 1$) is given by

$$G = \frac{1230\,(2.97 - e)^2}{1 + e}\,\frac{(CI + 25)}{500} \qquad \text{(EQ 28)}$$

where

CI = measured cone index

Currently Bekker's equation is used to get an approximate value of CI based on the properties stored in the database for an average penetration depth of 6 inches.

The void ratio, e, is given by

$$e = G_S \frac{\gamma_W}{\gamma_d} - 1 \qquad \text{(EQ 29)}$$

where

γ_w = unit weight of water = 62.4 lb/ft^3

γ_d = dry unit weight of soil

G_s = specific gravity of soil, $2.65 < G_S < 2.69$

The following tabulated values of e are used based on the soil type

TABLE 1. Void Ratios (e)*

SOIL TYPE	SW	SP	SM	SC	ML	MH	CL	CH	OL	OH
VOID RATIO	0.75	0.8	1.05	0.9	0.95	1.0	1.05	1.2	1.5	2.0

*. from [WES88a]

2.6.1.4 Motion Resistance

Motion resistance results from compaction, mechanical rolling resistance, vegetation, and other factors. The soil compaction, or the bulldozing resistance, occurs on the front or back inclined face of the track depending on the direction of motion. Since no comprehensive information is available to compute compaction resistance using VCI or CI, their effects are not included in the current implementation. The rolling resistance is a function of the normal force. The rolling resistance of a frictional surface is defined as,

$$R = \mu \text{ Normal Force} \qquad \text{(EQ 30)}$$

2.6.1.5 Compaction Resistance

The compaction resistance R_c is calculated as the work done in compacting the soil to a depth of z_o, where z_o is the sinkage. Since no relation to determine sinkage using VCI and RCI was available from WES, Bekker's pressure-sinkage relation is used [Bekker69].

$$Z_o = \left(\frac{p}{\frac{k_c}{b} + k_\phi} \right)^{1/n} \qquad \text{(EQ 31)}$$

pressure, $p = \frac{W}{bl}$, b and l are the track width and length respectively,

k_c = Cohesive modulus

$k\varphi$= Frictional modulus

n = Exponent of sinkage

The compaction resistance R_c is given by Bekker as,

$$Rc = \frac{1}{\{ (n+1)\, b^{1/n} (\frac{k_x}{b} + k_\phi)^{1/n} \}} (\frac{W}{l})^{(\frac{n+1}{n})} \qquad \text{(EQ 32)}$$

2.6.1.6 Normal Pressure

The Normal pressure is a sum of the components of the vehicle gross weight, inertia force,balance of vertical stresses and their moments and track tensions [WES 86, Baladi 78]

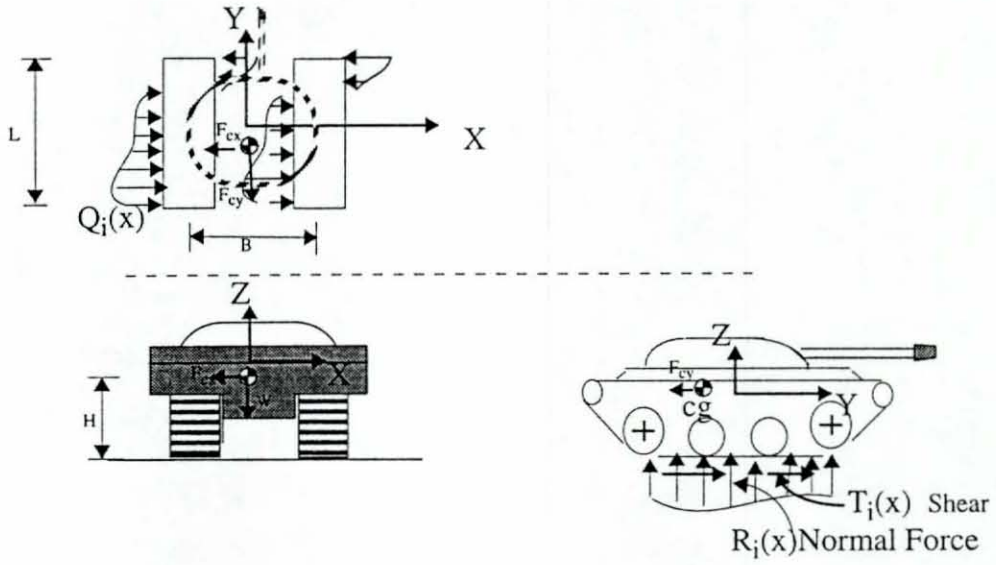


FIGURE 8: Distribution of forces on a tracked vehicle

The normal stresses R_i (Y) on the outer and inner track at a distance y is given by[WES86],

$$R_1(y) = \frac{W}{(dL^2)}\{\frac{1}{2} + 6yc_y - \frac{h}{b}\frac{F_{c_x}}{W} - 6hy\frac{F_{c_y}}{W} + dL^2\frac{N_1(y)}{W}\} \quad \text{(EQ 33)}$$

where

h = height of cg

b = centerline distance between the tracks

F_{cx},F_{cy} = Inertia Forces in the x & y directions

N_1 (y) = track tension at a distance x on the track

If the slope of the terrain, η is taken into consideration and the initial orientation the same as the yaw angle, ω, the normal stress is given by,

(EQ 34)

$$R_1(y) = \{\frac{\cos\eta}{2} + 6yc_y\cos\eta - \frac{h}{b}\left[\frac{F_{c_x}}{W} - \sin\eta\sin\omega\right] - 6hy\left[\frac{F_{c_y}}{W} + \sin\eta\cos\omega\right] + dL^2\frac{N_1(y)}{W}\}$$

The above expression is nondimensionalised and generalized for each track. The weight distribution is computed over each contact point and is a function of the slope of the terrain, the vehicle tilt owing to soil compaction and the independent suspension effects.

At high slips the shear stress becomes significant and hence the track tension which is a function of shear stress. An iterative scheme is required to compute the track tensions so as to satisfy the moment equilibrium relations at each point of analysis. The contributions of track tension towards the normal stresses are not included in the current implementation.

2.6.1.7 Rolling Motion Resistance

The rolling motion resistance on a frictional surface is given by

$$F = W \tan \Phi \, f(\text{Slip}) \qquad \text{(EQ 35)}$$

or

$$F = \mu \, W \, f(\text{slip}) \qquad \text{(EQ 36)}$$

For off road mobility, the WES empirical relations [WES92c] are used to compute μ for tracks riding on frictional soil types. For fine grained soil, the coefficient of motion resistance μ is computed as follows,

$$\mu_{fg} = \frac{0.045 + 2.3075}{RCI - VCI + 6.5} \qquad \text{(EQ 37)}$$

For soil types 0 & 1 it is assumed as 0.075. [WES92c]. Once the coefficient of motion resistance is computed, the motion resistance is given by,

$$F = \mu \text{ Normal Force} \qquad \text{(EQ 38)}$$

2.6.1.8 Tractive Force

The traction for a tracked vehicle is determined by the ability of the soil mass confined between the grousers to withstand shear force or, in other words, the ability of the grouser confined mass not to separate from the base soil. The friction between the track surface and the soil also aids in developing the thrust, as the wheels ride over the tracks. Theoretically the tractive effort in a tracked vehicle is developed by the shearing action of the track across the grouser tips, vertical surfaces and the soil surface. It is expressed as the integral of the shear stresses over the hypothetical contact area at the sliding surfaces. They could be expressed as,

$$F_h = \int_0^{l_t} \tau \cos\alpha \, dl \qquad \text{(EQ 39)}$$

and

$$F_v = \int_0^{l_t} \tau_v \sin\alpha dl \qquad \text{(EQ 40)}$$

where τ and τ_v are the shear stresses along the horizontal and vertical surfaces and α is the angle the track element makes with the horizontal. Tractive force is an important aspect in designing vehicle power requirements, traction devices and in determining mobility. Insufficient traction causes immobilization. The ultimate shearing resistance between the track soil interface for a cohesive soil is given by

$$\tau = c + \sigma_n \tan\phi \qquad \text{(EQ 41)}$$

where c is the soil cohesion, φ the friction angle and σ_n the normal stress. Bekker -Janosi's equation[Bekker69] for shear stress distribution as a function of slip is derived based on the solution of damped harmonic oscillator and is given by,

$$\tau = (C + \sigma_n \tan\phi)\ (1 - e^{-\Delta/K}) \qquad \text{(EQ 42)}$$

where K = Slip coefficient,

Δ = Shear deformation.

The hyperbolic shear stress-shear displacements relations used by WES model [Kondner63] is implemented here. The shear stress is given by,

$$\tau = \frac{G\ \tau_M\ \Delta}{\tau_M + G|\Delta|} \qquad \text{(EQ 43)}$$

where G = Shearing modulus,

τ_M = Maximum shear stress of the soil.

In the simplified WES model there is no method to directly compute the available traction on any given soil. WES has empirical results for the drawbar pull and the tow resistance. The sum, i.e. Drawbar pull + Motion resistance gives the traction available. It is not possible to use the above definition of tractive force for man-in-the-loop simulations. The engine power available etc. cannot be extracted. Instead, the theoretical definition of shear stress is used in computing the maximum available tractive force in the current implementation. The longitudinal shear stress [WES86] is given by,

$$T_i(y) = \frac{W\mu\delta_i}{L^2}\left(\frac{\left[c + \frac{r_i(y)}{d}\right]\tan\phi + C_d\left[1 - e^{(-\lambda\dot{\delta}_i)}\right]}{\mu|\delta| + \left[c + \frac{r_i(y)}{d}\right]\tan\phi + C_d\left[1 - e^{(-\lambda\dot{\delta}_i)}\right]}\right)\cos\gamma \qquad \text{(EQ 44)}$$

and the lateral shear stress is given by,

$$Q_i(y) = \frac{W\mu\delta_i}{L^2}\left\{\frac{\left[c+\frac{r_i(y)}{d}\right]\tan\phi + C_d\left[1-e^{(-\lambda\dot{\delta}_i)}\right]}{\mu|\delta| + \left[c+\frac{r_i(y)}{d}\right]\tan\phi + C_d\left[1-e^{(-\lambda\dot{\delta}_i)}\right]}\right\}\sin\gamma \qquad \text{(EQ 45)}$$

where

$r_i(y) = dL^2 R_i(y)/W$ = Nondimensionalised (ND) normal stress

$\delta i = \Delta\iota/L$ = (ND) Slip distance

$= \frac{\dot{\Delta}}{L}$ = (ND) Slip Veloci

$\mu = GL^3/W$ = (ND) Rigidity Modulus

$\lambda = \Lambda L$ = Soil attribute

$c = CL^2/W$ = (ND) Cohesive Strength

$c_d = C_d L^2/W$ = (ND) Dynamic Cohesive Strength

γ_t = Slip Angle

2.6.1.9 Slip

The shear stress at a point on the track is a function of normal stress and shear displacement, which is measured from the point on the track-terrain interface where shearing begins. The shear displacement is found from the slip velocity, Vs. The slip velocity of a particular point on the track is defined as the difference of vehicle velocity and the horizontal component of the track velocity at the point in consideration. The track velocity under different wheels could be different owing to the terrain, independent suspension and also due to the track flexibility. The slip velocity can be expressed as,

$$V_s = V_{rim} - V \cos \alpha \qquad \text{(EQ 46)}$$

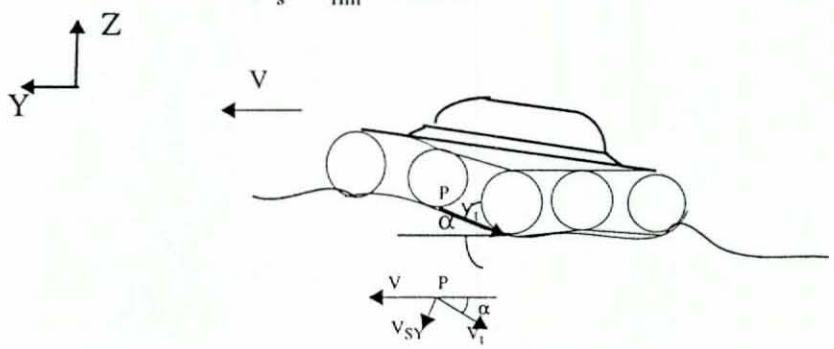


FIGURE 9: Slip velocity in a tracked vehicle

where V_{rim} is the tangential rim velocity and V is the vehicle velocity and α the trim angle.

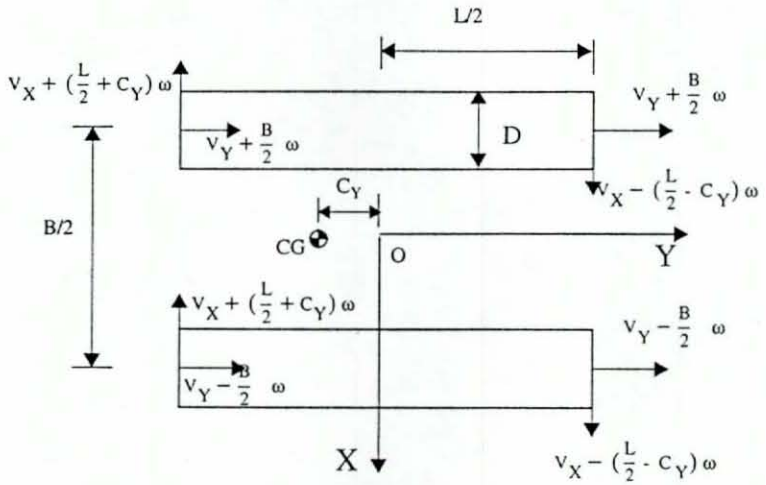


FIGURE 10: Track velocity of arbitrary point on the hull

With zero trim angle and including steering effects, we have the slip velocity resolved in the y direction given by,

$$V_{sy} = Vtrack - (V_Y + (b/2)\,\omega) \qquad \text{(EQ 47)}$$

where the second term on the right hand side is the velocity of the outer track in the direction of track(V_Y = Component of Vehicle velocity in the y direction), b the distance

from geometric center to center line of the track, and ω the rate of change of yaw angle of the vehicle, V_{track} = Rolling radius * ω_{rim}.

$$V_{sx} = \omega (X - C_x) \quad \text{(EQ 48)}$$

The resultant slip velocity is given by,

$$V_s = \sqrt{V_{sx}^2 + V_{sy}^2} \quad \text{(EQ 49)}$$

The shear displacement or the slip displacement is given by

$$\Delta = \int_o^t V_s dt \quad \text{(EQ 50)}$$

where t is the time taken to traverse between initial point of shearing and the point in consideration, with the current velocity in the y direction, V_Y. i.e.

$$t = (l/2 - Y)/ Vtrack \quad \text{(EQ 51)}$$

for points in front of the geometric center.

2.6.2 Equations of Motion

The equations of motion are obtained by applying Newtons laws of motion and is given by,

$$\text{Tractive Force - Motion Resistance} = M\,a \quad \text{(EQ 52)}$$

Using the definitions of tractive force, motion resistance described earlier in the above equation, we have,

$$\int_{-\frac{1}{2}}^{\frac{1}{2}} [t_1(y) + t_2(y)]\,dy - f \int_{-\frac{1}{2}}^{\frac{1}{2}} [r_1(y) + r_2(y)]\,dy = f_{cy} + \sin\eta\cos\omega \quad \text{(EQ 53)}$$

$$\int_{-\frac{1}{2}}^{\frac{1}{2}} [q_1(y) + q_2(y)]\,dy = f_{cx} - \sin\eta\sin\omega \quad \text{(EQ 54)}$$

$$\int_{-\frac{1}{2}}^{\frac{1}{2}} [q_1(y) + q_2(y)]\,(y - cy)\,dy + \frac{b}{2}\int_{-\frac{1}{2}}^{\frac{1}{2}} ([t_2(y) - t_1(y)]\,dy) + \frac{b}{2} f \int_{-\frac{1}{2}}^{\frac{1}{2}} r_1(y) + r_2(y)\,dy = \frac{I_z}{LW}\frac{d\omega}{dt} \qquad \text{(EQ 55)}$$

where,

f = Coefficient of rolling resistance,

I_z = Moment of Inertia of the vehicle about the z axis passing-

-through c.g.,

f_{cy}, f_{cx} = (ND) Inertia forces in y,x directions,

= F_{cy}/W, F_{cx}/W respectively and W is the weight of the-

-vehicle,

t_1 (y), t_2(y) = (ND)Longitudinal shear stresses for the outer and-

-inner track,

$= \frac{dl^2}{W} T_1(y), \frac{dl^2}{W} T_2(y)$ respectively,

q_1(y), q2(y) = (ND) Transverse shear stresses for the outer and-

-inner track,

$= \frac{dl^2}{W} Q_1(y), \frac{dl^2}{W} Q_2(y)$ respectively,

The other three equations for additional degrees of freedom, i.e. roll, pitch and motion in Z direction remain the same as the other mobility model.

2.6.3 Derivations for real-time Performance

The above set of equations of motion are integrated to obtain the position and velocity of the vehicle. The responses of man in the loop simulation are integrated into the equations of motion, while driving the vehicle on a terrain. Some of the iterative schemes are not taken into consideration for the initial real-time implementation (e.g., track tension). Runge-Kutta second and fourth order and Euler integration schemes were used to integrate the equations of motion depending on the stability, accuracy and speed requirements [Nakamura91, Press91].

2.6.4 Limitations of the Model

Plate sinkage equations will have to be used to find out the sinkage of the vehicle. It is not clear how VCI and CI could be used in computing the sinkage. Additional time is spent in

recomputing the cone index from the available soil properties in the database. Forces arising due to the independent suspensions are not completely modeled. Transmission and the vehicle performance curves are taken from comparable vehicles.

2.7 Investigation of Additional real-time Mobility Models - Bekker/Wong Tractive Effort Algorithms

As discussed in Section 2.2, the principal requirement for a dynamically correct vehicle simulator is the ability to determine what external forces and moments are applied to the vehicle. These forces and moments provide the vehicle's propulsion and steering, as well as support it's weight. Without these terms, the realism advantages of dynamically correct modeling cannot be obtained. This section discusses several relationships and algorithms that are designed to predict these forces and moments. These algorithms were developed by Dr. M.G. Bekker, Dr. J.Y. Wong, and others.

2.7.1 Theoretical Foundations

The theoretical principles (as opposed to empirical principals developed at WES) that govern the interactions between tracked vehicles and underlying terrain were first organized and developed in the 1950's by Dr. M.G. Bekker of the University of Michigan and the Stevens Institute [Bekker58, Bekker60, Bekker69]. Dr. Bekker analyzed the mechanics of transverse interactions in order to determine how propulsive forces and moments are generated by tracked vehicles. By studying how ground pressure and shear stress are distributed in the underlying terrain, he was able to develop standard integrals for modeling different types of transverse interfaces. These integrals are very flexible and can be easily modified to suit a particular configuration.

Since the 1970's, Dr. J.Y. Wong of the Ottawa Transportation Institute has developed these integrals, along with other relationships compiled by Dr. Bekker and others, into computer algorithms for use in evaluating, designing, and controlling tracked vehicles [Wong78, Wong89]. The primary input to the resulting equations is the slippage of the vehicle's tracks over the terrain. This slip is defined as the difference between the actual speed of the vehicle and the speed of the track. The mathematical conditions for defining slip are as follows:

- IF actual speed > zero THEN
- IF theoretical speed > actual speed THEN slip = 1.0 - (actual / theoretical)
- IF theoretical speed = actual speed THEN slip = 0.0
- IF theoretical speed < actual speed THEN slip = 1.0 - (theoretical / actual)
- IF actual speed = zero THEN
- IF theoretical speed = zero THEN slip = 0.0
- ELSE slip = 1.0
- IF actual speed < zero THEN
- Use the same relations as the first case, but reverse the tests

Note: The direction of the slip must be stored for the traction calculations

The works of Wong and Bekker, as well as of others quoted in their books, forms the theoretical basis for the tractive effort algorithms used by the Tracked Vehicle Simulator. These tractive and resistive forces and moments (generated by the track-terrain interface) are used in the dynamics algorithms as external forces and moments acting on the vehicle. This permits a standardized dynamics simulator, with all relevant physical realism, to be used to model a track-driven ground vehicle.

2.7.2 Derivations for Real-Time Performance

As mentioned in the prior section, the Bekker/Wong trafficability algorithms are based on analysis of shear-stress and pressure distributions, as well as the actual contact conditions through the transverse interfaces. The standard integrals and relationships that are used by these equations are generic in nature. That is, these relationships are sufficiently flexible that they can be configured for different specific vehicles and different levels of fidelity. These equations deal with two separate types of interactions. The first ones deal with transverse effects; i.e., forces that generate for-aft and sideways motion, as well as the moment that changes heading. The other relationships deal with perpendicular effects; i.e. the moments that generate pitching and rolling as well as the force that supports the vehicle's weight. The equations are as listed below.

for the transverse effects,

$$F_{MAX} = Ac + W\tan\phi \qquad \text{(EQ 56)}$$

$$K_{RATIO} = 1 - \left(\frac{K}{iL}\right)\left(1 - e^{-\left(\frac{iL}{K}\right)}\right) \qquad \text{(EQ 57)}$$

$$F_{ACTUAL} = F_{MAX}K_{RATIC} \qquad \text{(EQ 58)}$$

for the perpendicular effects,

$$Load = \left(\frac{W}{L}\right)^{\left(\frac{n+1}{n}\right)} \qquad \text{(EQ 59)}$$

$$Strength = (k_C + k_\phi B)^{\left(\frac{1}{n}\right)} \qquad \text{(EQ 60)}$$

$$Drag = \left(\frac{1}{n+1}\right)\frac{Load}{Strength} \qquad \text{(EQ 61)}$$

where,

F_{MAX} is the maximum possible force transmissible (in newtons)

F_{ACTUAL} is the actual force transmitted through the interface (in newtons)

K_{RATIO} is the ratio between the actual force and the maximum possible force

Load is an expression of the loading of the ground

Strength is an expression for the level of loading that the ground can take

Drag is the resistive force applied by plowing vehicle tracks into the ground

B is the width of the interface (in meters)

L is the length of the interface (in meters)

A is the area of the interface (in meters squared)

W is the force transversely applied to the interface (in newtons)

i is the slip at the interface (unitless)

c is the soil cohesion (newtons per meters squared)

ϕ is the angle of shearing resistance (in radians)

μ is the tangent of ϕ (unitless), a storage variable

K is the bulk modulus (in meters)

k_C is the cohesive modulus

k_ϕ is the frictional modulus

n is the exponent of soil deformation (unitless)

The second set of equations also deal with support forces and track sinkage. However, since vehicle tracks had not been implemented in the Dynamic Terrain Testbed at the time this work was completed and, thus, no depression is made in the simulated terrain by the vehicle, these relationships were not utilized like the transverse equations.

Because of the high performance and fidelity requirements of the Tracked Vehicle Simulator, more complicated versions of the basic algorithms are utilized. These implemented algorithms to model the terrain underneath each road-wheel of each track. Doing so provides accurate prediction of the major forces concentrated on the vehicle, as most loads are generated underneath the road wheels.

2.7.3 Assumptions and Limitations of the Model

The Bekker/Wong algorithms function fairly efficiently and robustly. The only drawback to their application is very high sensitivity to low track speeds. When the difference between the theoretical track speed and the actual speed is within a couple of percentage points, the forces and moments generated through the contact interface can change drastically with modest changes in the relative speed differential. Around zero slip, the change is very rapid. Correctly modeling the track/terrain interface therefore requires more attention to detail in this situation. With the dynamics simulation running on the same processor as the image generator, only a few extra calls per frame can be made. Therefore, a safety routine is in place that performs additional calculations as required. This test watches for transitions between positive and negative slip. When it detects such a transition, it performs two sets of calculations: one set directly before the transition and

the other one set directly after. This provides increased stability in the calculations by providing higher fidelity around the most transient value of slip, zero. This case is very important because it represents when the vehicle is parked or is coming to a rest.

The primary assumption of this development is the pressure distribution underneath the vehicle's tracks. For the purposes of this model, the contact pressure is assumed to be concentrated underneath the tracked vehicle's road wheels. The Bekker integrals are customized for this case. This is actually better than the simpler integrals that Bekker and Wong use, however, they were not concerned with real-time simulation. While solving several integrals per time loop may seem time consuming, the increased accuracy that results allows the entire system to run less often. This generates a far greater time savings than the increased number of calculations per iteration.

The Bekker/Wong mobility model also makes several assumptions to allow for on the fly solution. First a standard form of the traction equations is pre-integrated into closed form. In order to make these equations as generic as possible, the closed form assumes the basic nature tracked vehicle motion with symmetric tracks (i.e. the tracks are identical and are set equidistant from the center of the vehicle). The integration was performed in a generic nature so that a different number of equations can be used for vehicles with a different number of road wheels per track.

3.0 Software

3.1 Software System Overview - The Tracked Vehicle Simulator (TVS)

The Tracked Vehicle Simulator (TVS) is written in C++ on Silicon Graphics workstations. The Silicon Graphics (SGI) system was chosen for its graphics capabilities and for the baseline capability provided by the Performer and FORMS software libraries [SGI92a, SGI92b, Overmars91]. Both of these Application Programmer Interface (API) tool kits are discussed further in this document.

To reduce development time, software developed for other portions of the DT Testbed were reused whenever possible. For instance, the entity manager and model manager from. The DT Visualization System, also known as BwanaVision (BV), were included in the TVS. Also included were Performer-based objects (fog, earth/sky model), the dynamic terrain generator, and the BV explosion routines. Incorporation of these and other software modules was facilitated by the use of object-oriented design methods in TVS and throughout the DT Testbed software. However, to aid integration into the DT Testbed, the entity manager was modified to use entity services [VSL94]. Formerly, BV only communicated through DIP (DIS Interface Process), a software package written by the Naval Training Systems Center (NTSC) for communications with other DIS simulations. Explanation of these software modules is beyond the scope of this document and can be referenced in the appropriate DT Testbed project documentation [Buckley93, VSL94].

3.1.1 Module Design

TVS is divided into modules, referred to as managers, which handle the different simulator functions. These managers are the Simulation Manager (Sim_Mgr), Image Generation Manager (IG_Mgr), Vehicle Dynamics Manager (VDYN_Mgr), Environment Manager (ENV_Mgr), Entity Manager (Ent_Mgr), and the Graphical User Interface Manager (GUI_Mgr). Other supporting modules include the Dynamic Terrain Generator (DTGen) and the Entity Services Entity Manager (ESERV_EntityManager). The interaction of these modules is shown in Figure 11 and descriptions of the modules are provided in the following sections.

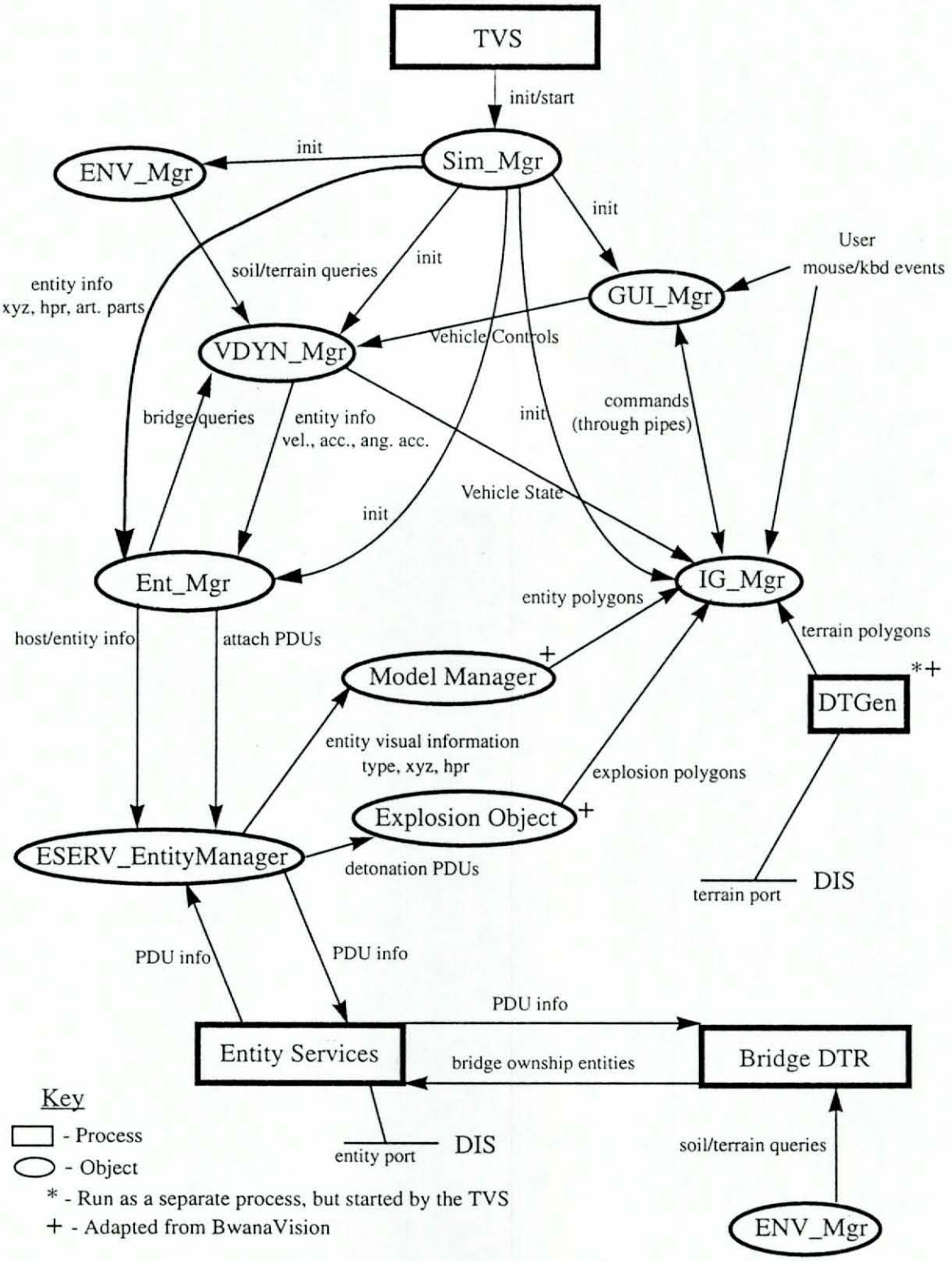


FIGURE 11: Data Flow and Object Relationships of the Tracked Vehicle Simulator (TVS)

3.1.1.1 SIM_Mgr - Simulation Manager

The Simulation Manager controls the simulation and is the only object found in the main source code. It constructs and initializes the other managers and deletes them upon the end of the simulation loop. It also provides the functions necessary for debug printing and fatal error handling. It handles hardware and process specific details, such as initializing a fast timer if the user ID is *super-user*, and locking the process and data blocks into memory for faster performance (i.e., no virtual memory swapping).

3.1.1.2 IG_Mgr - Image Generator Manager

Rendering of the scene graph is accomplished by the IG_Mgr. For this purpose, we have selected Performer, an Application Programmer Interface (API) tool-kit written by SGI for high performance rendering in simulation based applications [SGI92a, SGI92b]. The *scene graph* refers to the geometry tree created by the Performer routines. Each call to load a model file adds polygons, or polygon meshes, to the geometry tree. The IG_Mgr is responsible for initializing shared memory spaces for use by other managers, starting the dynamic terrain object (or loading the terrain file), loading the vehicle model, and opening up the process command pipes which reside in the shared memory just created. More information on communication with the dynamic terrain system can be found in a later section of this report.

The shared memory structure created in the TVS is handled completely by the Performer libraries. Shared memory access requires a lot of tricky manipulation with semaphore control or file lock access. For Performer to configure shared memory properly, the data must be allocated between the Performer functions *pfInit*() and *pfConfig*(), which are called within the IG_Mgr. Therefore, the Sim_Mgr cannot allocate and initialize the shared memory structure.

3.1.1.3 VDYN_Mgr - Vehicle Dynamics Manager

This manager implements the real-time dynamics of the vehicle to be simulated. The IG_Mgr maintains the current frame rate of the system and updates the VDYN_Mgr on those frame rate boundaries (or some multiple thereof, if the vehicle dynamics model needs to run faster), and passes the time step (Δt) through the *update*() method. The VDYN_Mgr implements its own objects for such functions as modeling different subsystems or controlling articulated parts of the vehicle (e.g., a bulldozer blade or the AVLB bridging mechanism).

3.1.1.4 ENV_Mgr - Environment Manager

The Environment Manager handles the elevation, slope, and other terrain property queries for the Vehicle Dynamics Manager. Terrain property queries related to the terrain geometry involve intersection testing with the Performer geometry scene. These return the Z-values and normal vectors for the given vehicle contact points. This information is used by the VDYN_Mgr for terrain following. The ENV_Mgr also provides the capability to query for the underlying soil properties of the terrain at the same contact points. The DT generator, though not connected with the ENV_Mgr, produces all the polygons on the

terrain geometry scene.

3.1.1.5 ENT_Mgr - Entity Manager

This manager communicates with the BV Entity Manager object (ESERV_EntityManager) as shown in Figure 11 and Figure 12. The ENT_Mgr is responsible for handling all Distributed Interactive Simulation (DIS) Protocol Description Units (PDUs) received from the network [IST93a, IST93b]. It maintains a list of the currently active entities and generates their visual geometry for display by the IG_Mgr. It communicates with a separate process, Entity Services, which provides dead reckoning of the entities and their articulated parts. As shown in Figure 12, the DIP process served the role of Entity Services in Bwanavision for the version of BV developed for the Naval Training Systems Center [Buckley93]. The BV Entity Manager also provides the capability to send out ownship entity state PDUs for the host vehicle.

The Ent_Mgr also receives packets from the net other than Entity State PDUs, such as Detonation, Detach/Attach (new PDUs created for the bridging vehicle), and Fire PDUs. For Detonation PDUs, the Ent_Mgr instantiates a new explosion object to create a visual explosion in the IG. This object handles its own display through Performer pre-draw and post-draw function callbacks. When the explosion is finished, the object resets itself to initialization state and therefore no longer sends draw commands to the active visualization window. For more information on BV, function callbacks, or Performer, please refer to the respective manuals [Buckley93, SGI92a, SGI92b].

All dynamic terrain PDUs are passed along a separate ethernet port. This allows a system without Dynamic Terrain capability to operate within the DT Testbed provided that it does not need to visualize the changing terrain. The DT generator process, from BV, receives these packets and incorporates any changes in the terrain scene directly by creating or destroying terrain polygons. No internal object of the TVS ever receives dynamic terrain changes. In fact, the TVS will not even be aware of these changes as the DT generator modifies the geometry scene directly.

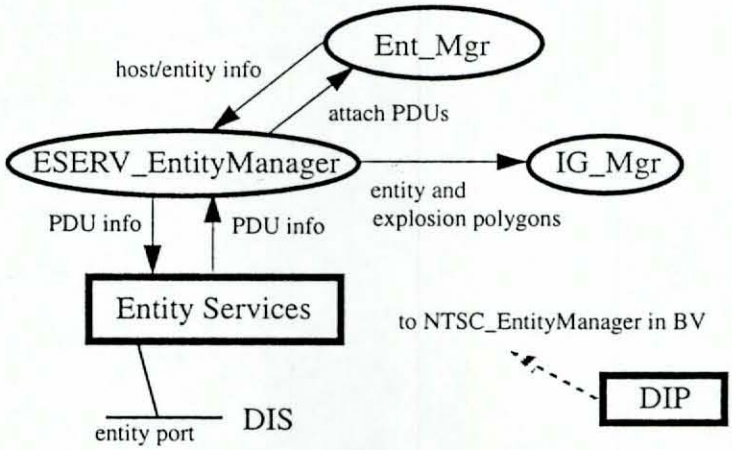


FIGURE 12: Application of the Bwanavision Entity Manager in TVS

3.1.1.6 GUI_Mgr - Graphical User Interface Manager

The GUI_Mgr is responsible for all communication between the application and the user (i.e., the vehicle driver). This interface is implemented with the aid of FORMS. FORMS is an API written for SGI workstations which allows for easy definition of a graphical user interface (GUI) [Overmars91]. This includes the ability to define methods to receive input from the GUI and to display output from the application through the GUI. The interface can receive input from either the mouse, keyboard or both. It also provides centralized keyboard bindings to tie key strokes in the IG visualization window to the same commands in the GUI window. That is, when a key is pressed when the cursor is in the IG visualization window, it will perform the same function as if pressed in while the cursor is in the GUI portion of the window. Wherever a key is read from the queue, the IG_Mgr command pipe is sent the corresponding command. Most of the keyboard commands reference IG_Mgr specific functions.

3.1.2 Communication Between Modules

To reduce the amount of dependency between the manager objects and allow easy extensibility, additional objects were created to pass large amounts of data between the managers as well as flexible methods to pass commands between the managers. These new objects include a Soil Sample object, a Vehicle State object, and a Vehicle Controls object. The methods referred to are implemented through command pipes. These methods and objects are described in the following sections.

3.1.2.1 Soil Samples

To transfer different types of terrain queries between the ENV_Mgr and the VDYN_Mgr, a soil sample object was designed. This object encapsulates the necessary data for intersection points, elevations, surface normals, and soil properties.

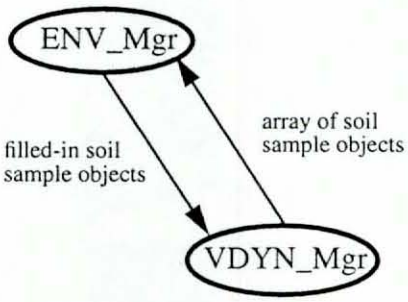


FIGURE 13: Soil_Sample Passing

Figure 13 shows the use of the Soil_Sample object. The array of Soil_Sample objects is created in the VDYN_Mgr and passed to the ENV_Mgr. The Env_Mgr then fills the soil sample array with the requested properties. However, this array can never go beyond the maximum number of soil samples set in the ENV_Mgr at initialization of the ENV_Mgr. The VDYN_Mgr, just before calling the soil query functions, must supply the world coordinate positions of the segments for intersection and their corresponding normalized

direction vectors (i.e., the soil sample locations). For more information on this interaction, please refer to the sections on the ENV_Mgr and the VDYN_Mgr.

3.1.2.2 Vehicle State

A hierarchy of objects was created for passing the current vehicle state between the VDYN_Mgr, IG_Mgr, and the GUI_Mgr as shown in Figure 14. It consists of a top-level object with the basic vehicle state, such as position, velocity, acceleration, and other variables defined by the Vehicle Dynamics manager. From the Veh_State object, a set of sub-classes are defined to supply additional state information for specific vehicle types supported by the TVS. Due to polymorphism (a property of object oriented languages), these derived objects inherit all of the variables and methods of the parent Vehicle State object. Currently, the vehicle types supported are the AVLB and the M9 bulldozer armored utility vehicles. Additional state information for the AVLB would include hinge angles of the deployable folding bridge as well as rotational rates.

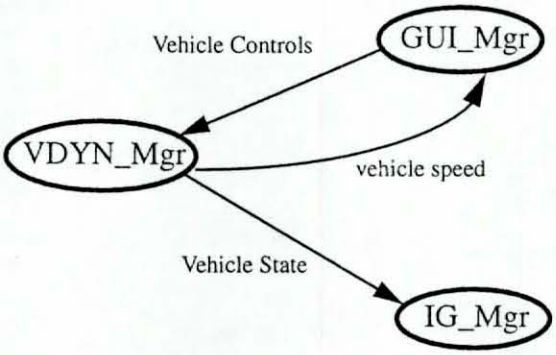


FIGURE 14: Vehicle State and Vehicle Controls Passing

3.1.2.3 Veh_Cntrls - Vehicle Controls

To transfer control information from the GUI_Mgr to the VDYN_Mgr (see Figure 14), the Veh_Cntrls class was created. This class handles such information as the steering controls, braking, and transmission controls. More specific vehicle functions are passed through a sub-class of Veh_Cntrls, to either the AVLB or the M9. This hierarchical structure is similar to the vehicle state objects.

3.1.2.4 Command Pipes

To facilitate the passing of commands between the managers that would allow for extensions of number and type of commands, the pipe function provided by the UNIX standard libraries was used. For more information, on the pipe function see *pipe* in the C reference manual, from the header file *unistd.h*. The UNIX manual pages describe "pipe" as:

```
...pipe creates an I/O mechanism called a pipe and returns
      two file descriptors, fildes[0] and fildes[1].
```

```
        Fildes[0] is opened for reading and fildes[1] is
        opened for writing.

Up to PIPE_BUF (defined in limits.h) are guaranteed to be
        written atomically. Up to PIPE_MAX (defined in
        limits.h) bytes of data are buffered by the pipe
        before the writing process is blocked. A read only
        file descriptor fildes[0] accesses the data written to
        fildes[1] on a first-in-first-out (FIFO) basis.[1]
```

A command pipe is created for each simulation manager which needs to communicate with another manager. The managers which use pipes are the IG_Mgr, GUI_Mgr, VDYN_Mgr, and ENV_Mgr. Using this method, any manager can communicate (send commands) with any other manager's command pipe, without worrying about inaccessible process or object space. The commands will then be handled by the appropriate object manager when that manager gets updated. For example, when the GUI_Mgr receives a command to turn on fog, it writes a "fog on" command to the IG_Mgr's input command pipe (i.e., the read file), which the IG_Mgr will process on the next frame update. If necessary, the IG_Mgr could transfer commands to another manager, such as responding to the GUI_Mgr if a command was not processed (e.g., due to an error condition) to allow the GUI_Mgr to reflect the correct status of the system to the user.

The command pipe structure aids in parallelization in the future. In theory, each manager may reside in a completely separate process. Otherwise, there needs to exist some method with which to communicate between these managers that does not rely on data access (i.e, global memory). Another method to communicate is shared memory, but this would become difficult after many commands have been sent between all the managers. Also, it may get backlogged, creating more problems such as running out of command space, if the programmer were not to account for a large backlog.

3.1.3 Communications with the DIS Network

The Entity Services interface program reads all DIS PDUs from the network related to entity activities. This is a stand alone process which runs in parallel with the TVS. However, Entity Services (ESERV) also possesses the ability to communicate with other processes on the same machine. For example, the AVLB Bridge DTR needs access to ESERV as well as the TVS. We can therefore run both programs on the same machine, with the ESERV running to provide them with PDU information. It is a characteristic of UNIX workstations that more than one program cannot bind to the same ethernet port. Therefore, only one program can accept information from the DIS network. This constitutes a principle reason that ESERV was created [VSL94].

ESERV provides enhanced capability and flexibility to the subscribing programs. The functions offered include remote entity approximation of ghost entities and dead reckoning of articulated parts. Therefore, this functionality need not be repeated per application, but only per processor or network node (see Figure 15). Furthermore, with

1. Man page on PIPE(2) from Silicon Graphics, Inc. IRIX Release 4.0.5

this service being provided on each processor/node, the applications have the flexibility of moving to any node where the processor can accommodate the load of that particular application. Entity Services is detailed further in upcoming documentation and is part of the DT Testbed development [VSL94].

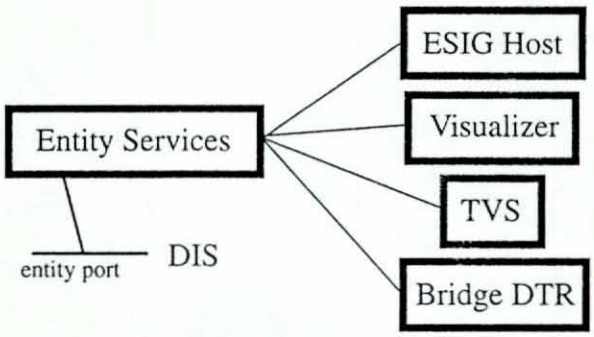


FIGURE 15: Multiple Processes Attached to Entity Services on a Single Machine

Because the Bridge DTR and the TVS are separate processes, they need to either communicate via shared memory, file, or some other method. The constraint to run both of these processes on the same machine appeared unreasonable since both may be processor intensive. Therefore, a different method for communication was chosen, namely by using a new DIS PDU called the Attach/Detach PDU. This PDU alerts other nodes on the net that a removable, articulated part (e.g., the bridge of the Armored Vehicle Launched Bridge) has been attached or detached from the entity sending the PDU. The actual format of the Attach/Detach PDU is not specific to the AVLB bridge, so it may be used for other entities which have removable, articulated parts. The following section of code is extracted from the Entity Services program. It shows the attach PDU as defined on our SG machines.

```
typedef struct {
PDUHeader header;
u_char isAttach; // if TRUE, attach -- if FALSE, detach
EntityID3 attachER_ID; // attachER becomes a child
EntityID3 attachEE_ID; // of attachEE
double worldX, worldY, worldZ;
float h, p, r;
float entityX, entityY, entityZ;
u_char numParts;
ArtMsg articulatedParts[MAX_ART_PARTS];
} AttachMsg;
```

The Bridge DTR utilizes the Attach/Detach PDU in the manner described below. When the Bridge DTR receives a Detach PDU, it creates its own bridge entity and it is given the entity ID from the PDU information to use for this new bridge. The PDU is sent out by the TVS. The Bridge DTR now sends out the bridge entity state PDUs and treats it like any other entity on the network. Also, the TVS knows to deactivate its bridge's visual model and bridge dynamics and to load the new visual model of the detached bridge when it receives the first bridge entity state PDU. Therefore, the bridge is now treated like any

other entity on the network. The AVLB can now react to its own bridge and to any deployed bridge in the same manner. It can, for instance, attach its tongue cylinder to another AVLB's bridge, as well as drive over it.

3.1.4 Communications with the Dynamic Terrain Testbed

Communications with the Dynamic Terrain Testbed occurs within the BV dynamic terrain manager object (DTMObj). The IG Manager initializes this object. The DTMObj forks a process which binds itself to the correct port for dynamic terrain ethernet traffic. The Database Services program, similar to Entity Services, but obviously for dynamic terrain, is not used by the DTMObj. Instead, a customized version of the database services code is included within it. This code was adapted to accommodate the high performance and frame synchronization requirements for BwanaVision. Therefore, the DTMObj reads DT PDUs directly from the network via the dynamic terrain port specific in the configuration file.

In the future, the DTMObj will be modified to use a separate copy of the Database Services. This will allow for other process which need access to dynamic terrain to run on the same machine. Please refer to BV documentation for more information on the DTMObj.

3.2 TVS Detailed Software Design

3.2.1 Data Transfer Objects

3.2.1.1 Veh_State - Vehicle State Object

The Vehicle State object contains general information such as acceleration, velocity, and normal vector. These attributes are common to all land-based entities and, as such, are kept in the top-level object of the hierarchy.

Here is an overview of the variables (i.e., attributes) contained in each Vehicle State object:

- world position - from the world origin to the bottom center of the vehicle
- normal vector - as it rest on the terrain
- direction vector (normalized) - for the front of the vehicle
- velocity vector - meters per second (m/s)
- acceleration vector - m/s^2
- speed - m/s
- orientation (heading, pitch, roll) - degrees

The following is a list of the public methods in the Vehicle State object. These methods either set or read an attribute of the vehicle state. A method not prefixed by "set_" is a read method. In some instances, more than one method will be provided to do a set or read. Typically, this allows a calling routine or object to extract or provide data in multiple

forms (e.g., radians or degrees, vector components or vector object). For most of the methods, the name of the method describes its function. Where the function is not obvious, a description is provided. The methods are as follows:

```
void set_position(float x, float y, float z);
void set_position(floatVector in_position);
floatVector position();
void set_normal(float x, float y, float z);
void set_normal(floatVector in_normal);
floatVector normal();
void set_direction(float x, float y, float z);
void set_direction(floatVector in_direction);
floatVector direction();
void set_speed(float in_speed);
float speed();
void set_velocity(float x, float y, float z);
void set_velocity(floatVector in_velocity);
floatVector velocity();
void set_acceleration(float x, float y, float z);
void set_acceleration(floatVector in_accel);
floatVector acceleration();
void set_orientation_in_radians(float in_heading, float
      in_pitch, float in_roll);
void set_heading_radians(float in_heading);
void set_pitch_radians(float in_pitch);
void set_roll_radians(float in_roll);
float heading_radians();
float pitch_radians();
float roll_radians();
void set_orientation_in_degrees(float in_heading, float
      in_pitch, float in_roll);
void set_heading_degrees(float in_heading);
void set_pitch_degrees(float in_pitch);
void set_roll_degrees(float in_roll);
float heading_degrees();
float pitch_degrees();
float roll_degrees();
```

The following set of methods define the current viewing positions for the user interface:

```
floatVector front_view_position();
floatVector left_view_position();
floatVector right_view_position();
floatVector rear_view_position();
floatVector front_view_direction();
floatVector left_view_direction();
floatVector right_view_direction();
floatVector rear_view_direction();
```

The following set of methods are used to manipulate status flags to determine when the vehicle has moved ("position_changed"), a part of the vehicle has moved ("components_changed"), if terrain effects on the vehicle are to be considered ("mobility"), or debugging statements are to be printed ("printOn"):

```
LogicalType position_changed();
void set_position_changed();
void clr_position_changed();
LogicalType components_changed();
void set_components_changed();
void clr_components_changed();
LogicalType mobility();
void set_mobility();
void clr_mobility();
void printOn();
```

3.2.1.2 AVLB State Object

The AVLB state object contains more specific information such as the transmission state and bridging state of all three bridging components. This object is a child of Veh_State and inherits attributes and methods from its parent. This means that AVLB_State has full access to all private and public members of Veh_State.

Here is an overview of the variables (i.e., attributes) contained in each AVLB State object:

- bridge component angles - degrees
- bridge attached flag - true if the bridge is still attached to the AVLB
- bridge locked flag - true if the locking pins are engaged on the tongue cylinder
- clutch engaged flag - true if the clutch has been engaged
- park locked flag - true if the park lock lever has been pulled
- bridge destroyed flag - true if the bridge has been destroyed, independent of the AVLB itself
- bridge direction vector - normalized alignment vector for the detached bridge; this is useful when trying to re-attach
- tongue direction vector - alignment vector for the tongue cylinder; this is also used when trying to re-attach
- bridge component angular velocity - radians per second (r/s)
- bridge position and orientation - this is only used when the bridge is a separate entity

The following is a list of the public methods in the AVLB State object. The first set of methods either set or read an angle of the joints used to attach the bridge to the AVLB. A method not prefixed by "set_" is a read method. In some instances, more than one method will be provided to do a set or read. Typically, this allows a calling routine or object to extract or provide data in multiple forms (e.g., radians or degrees, vector components or vector object):

```
void set_overhead_angle_degrees(float in_outrig_angle);
void set_overhead_angle_radians(float in_outrig_angle);
float overhead_angle_degrees();
float overhead_angle_radians();
void set_tongue_angle_degrees(float in_tongue_angle);
void set_tongue_angle_radians(float in_tongue_angle);
float tongue_angle_degrees();
float tongue_angle_radians();
```

```
void set_scissor_angle_degrees(float in_scissor_angle);
void set_scissor_angle_radians(float in_scissor_angle);
float scissor_angle_degrees();
float scissor_angle_radians();
void set_overhead_angle_rate_degrees(float
     in_outrig_angle_rate);
void set_overhead_angle_rate_radians(float
     in_outrig_angle_rate);
float overhead_angle_rate_degrees();
float overhead_angle_rate_radians();
void set_tongue_angle_rate_degrees(float
     in_tongue_angle_rate);
void set_tongue_angle_rate_radians(float
     in_tongue_angle_rate);
float tongue_angle_rate_degrees();
float tongue_angle_rate_radians();
void set_scissor_angle_rate_degrees(float
     in_scissor_angle_rate);
void set_scissor_angle_rate_radians(float
     in_scissor_angle_rate);
float scissor_angle_rate_degrees();
float scissor_angle_rate_radians();
```

The following methods manipulate state flags used in attaching the bridge, controlling other vehicle systems relevant to AVLB Bridge operation (i.e., clutch or parking brake), or determining if joint angles should change from operator inputs:

```
void set_bridge_attached();
 void clr_bridge_attached();
 LogicalType bridge_attached();
 void set_bridge_locked();
 void clr_bridge_locked();
 LogicalType bridge_locked();
 void set_clutch_engaged();
 void clr_clutch_engaged();
 LogicalType clutch_engaged();
 void set_park_locked();
 void clr_park_locked();
 LogicalType park_locked();
 void set_overhead_changed();
 void clr_overhead_changed();
 LogicalType overhead_changed();
 void set_tongue_scissor_changed();
 void clr_tongue_scissor_changed();
 LogicalType tongue_scissor_changed();
```

These methods are used to control bridge-vehicle interactions during the retrieval of a deployed bridge:

```
void set_bridge_destroyed();
void clr_bridge_destroyed();
LogicalType bridge_destroyed();
floatVector locking_point();
floatVector lockon_one_center();
```

```
floatVector lockon_two_center();
float lockon_one_radius();
float lockon_two_radius();
void set_locking_point(float x, float y, float z);
void set_lockon_one_center(float x, float y, float z);
void set_lockon_two_center(float x, float y, float z);
void set_lockon_one_radius(float r);
void set_lockon_two_radius(float r);
floatVector bridge_direction();
void set_bridge_direction(float x, float y, float z);
floatVector tongue_direction();
void set_tongue_direction(float x, float y, float z);
floatVector fulcrum_rotation();
void set_fulcrum_rotation(float x, float y, float z);
floatVector bridge1_rotation();
void set_bridge1_rotation(float x, float y, float z);
floatVector bridge2_rotation();
void set_bridge2_rotation(float x, float y, float z);
void set_bridge_orientation( float h, float p, float r );
void set_bridge_position( float x, float y, float z );
floatVector get_bridge_orientation( void );
floatVector get_bridge_position( void );

void printOn();
```

- prints debugging information

3.2.1.3 Veh_Cntrl - Vehicle Controls Object

This object reads inputs from the GUI_Mgr, translates these inputs into control values, and then transfers the control values to the VDYN_Mgr.

Here is an overview of the variables (i.e., attributes) contained in each vehicle controls object:

- transmission controls - park, neutral, low, high, reverse
- accelerator value - application specific range
- brake pedal value - application specific range
- steering value - application specific range

The translation of user input values into vehicle control values is an approach to decouple the dependency of the VDYN_Mgr from the GUI_Mgr by specifying these dependencies in a separate object. Where the phrase "application specific range" appears, the programmer must define the lower and upper bound values for the control mechanisms. A minimum and maximum value must be given for both the GUI_Mgr and the VDYN_Mgr. For example, the TVS defines the GUI accelerator values as 0.0 to 5.0 and the vehicle dynamics accelerator values according to vehicle specifications. This object performs the translation of values from the GUI to values within ranges anticipated by the vehicle dynamics model when its methods are accessed.

wing is a list of the public methods in the Vehicle Controls object:

```
void set_GUI_ranges(float accel_min, float accel_max, float
     brake_min, float brake_max, float steer_min, float
     steer_max);
```

- Sets the range of values the GUI_Mgr expects to read and write

```
void set_VDYN_ranges(float accel_min, float accel_max, float
     brake_min, float brake_max, float steer_min, float
     steer_max);
```

- Sets the range of values the VDYN_Mgr expects to read and write

lowing methods either read the vehicle controls or set them to a new value. These the accelerator, brakes, steering, and transmission:

```
oid set_accel_rate(float accel_rate);
float accel_rate();
oid apply_brakes(float brake_rate);
oid stop(); // full brakes
void full_throttle();
loat brake_rate();
oid maintain_speed();
float minimum_accel();
float minimum_brakes();
oid turn(float steering_angle);
oid go_straight();
float steering_position();
oid shift_to_park();
oid shift_to_neutral();
void shift_to_low();
oid shift_to_high();
oid shift_to_reverse();
TransmissionType in_gear();
LogicalType in_park();
ogicalType in_neutral();
ogicalType in_low();
LogicalType in_high();
ogicalType in_reverse();

void printOn();
```

- Output debugging information

AVLB_Cntrls - AVLB Controls Object

LB Controls object contains information specific to the AVLB while performing ne type of funcionality as the Vehicle Controls object. This object is a child of rl and inherits attributes and methods from its parent.

an overview of the variables (i.e., attributes) contained in each AVLB controls

- scissor cylinder lever - describes the state of this cylinder (BridgeCntrlType)
- tongue cylinder lever - describes the state of this cylinder (BridgeCntrlType)
- overhead cylinder lever - describes the state of this cylinder (BridgeCntrlType)
- ejection lever - true if the bridge has been ejected
- locking pins state - describes the current state of the locking pins lever

The type for the cylinder levers is defined as the following:

```
typedef enum BridgeCntrlType {EXTEND, CONTRACT, STOP};
```

The following is a list of the public methods in the AVLB Controls object. These methods control bridge deployment mechanisms and are self explanatory:

```
void extend_scissor_cyldr();
void contract_scissor_cyldr();
void stop_scissor_cyldr();
BridgeCntrlType get_scissor_cyldr_state();
void extend_tongue_cyldr();
void contract_tongue_cyldr();
void stop_tongue_cyldr();
BridgeCntrlType get_tongue_cyldr_state();
void extend_overhead_cyldr();
void contract_overhead_cyldr();
void stop_overhead_cyldr();
BridgeCntrlType get_overhead_cyldr_state();
void eject();
int get_eject();
void extend_lock_pins();
void contract_lock_pins();
BridgeCntrlType get_lock_pins_state();

void printOn();
```

- Output debugging information

3.2.1.5 Soil Sample Object

The Soil Sample object was developed to decouple the dependency of the Vehicle Dynamics Manager on the Environment Manger. It was realized early in the design that vehicle dynamics simulations such as TVS may have to support different models for terrain-vehicle interaction. It was also realized that these different models would require different types of soil properties. Therefore, it was decided to define a Soil Sample object that could be used to query the Environment Manager for soil properties instead of defining separate methods within the Environment Manager to extract these properties. This would reduce the amount of modifications required should a different terrain-vehicle interaction model be used.

The Soil Sample object contains the following attributes:

- cohesion of terrain soil (pressure)

- angle of shearing resistance (frictional property)
- bulk modulus of soil (length)
- cohesive modulus of deformation
- frictional modulus of deformation
- exponent of soil deformation
- surface normal - floatVector
- soil sample location - world coordinates
- valid flag - true if the intersection point for this soil sample was found
- intersection mask - contains the value for the intersection polygon, describing terrain, vehicle, or anything else

The following is a list of the public methods in the Soil Sample object. The first set of methods are used to set or read the sample location:

```
void take_at(float x, float y, float z);
void take_at(float x, float y);
float x();
float y();
float z();
floatVector xyz();
```

The following set of methods are used to set or read the elevation at the sample location or the surface normal and its components (snx, sny, snz):

```
void set_elev_n_surf_norm(float in_elev, float
     in_surf_norm_x,float in_surf_norm_y,float
     in_surf_norm_z);
void set_elev_n_surf_norm(float in_elev, floatVector
     in_surf_normal);
float elev();
floatVector surf_norm();
void set_surf_norm(float x, float y, float z);
float snx();
float sny();
float snz();
```

The next set of methods sets or reads the soil properties used in the mobility calculations:

```
void set_soil_properties(float in_c, float in_phi, float
     in_K, float in_Kc, float in_Kphi, float in_n);
float c();
float phi();
float K();
float Kc();
float Kphi();
float n();
```

The following methods are used to determine if valid soil properties are present in the soil sample object:

```
void set_valid();
 void clr_valid();
 LogicalType get_valid();

 void printOn();
```

- Outputs debugging information

3.2.1.6 floatVector Object

The *floatVector* object is used for three dimensional vector information. The surface normal, vehicle position, and other data are transferred via this vector object.

It was designed using a generic three-tuple header and source file. The specific variables in *floatVector* are macro defined into their respective names of *x,y,z* in the header file. The source file merely includes the definitions of the header file and the completely generic three-tuple source code. Macro instantiation for the three-tuple data type was used because our current version of C++ does not support templates, an easier name substitution method.

The following is a list of the public methods in the floatVector object. These methods are used to define basic vector operations:

```
void set(a3TUPLE_DATA_TYPE*);
void set(a3TUPLE_DATA_TYPE, a3TUPLE_DATA_TYPE,
     a3TUPLE_DATA_TYPE);
void set(a3TUPLE_AGGREGATE_STRUCT&);
a3TUPLE_DATA_TYPE a3TUPLE_ELEM_1_NAME();
a3TUPLE_DATA_TYPE a3TUPLE_ELEM_2_NAME();
a3TUPLE_DATA_TYPE a3TUPLE_ELEM_3_NAME();
a3TUPLE_AGGREGATE_STRUCT get();
NAME& operator=(NAME);
short operator == (NAME );
void operator += (NAME );
void operator -= (NAME );
void operator *= (a3TUPLE_DATA_TYPE );
NAME operator + (NAME );
NAME operator - (NAME );
a3TUPLE_DATA_TYPE magnitude();
void printOn();
a3TUPLE_DATA_TYPE *returnValues();
```

Macro substitution, as explained earlier, is used for the definition of this object. The file *floatVector.h++* contains the macro defines for the above names which are included from the generic header file, *3Tuple.gh++*.

3.2.2 Simulation Manager (Sim_Mgr) Object

For a list of all attributes and methods, please refer to the source code header file

Sim_Mgr.h++. The primary methods for this object are described below.

As stated previously, the Simulation Manager controls the simulation. Its responsibilities include initialization and deletion of the other managers and error handling. During initialization, the Sim_Mgr performs the following functions:

- creates and initializes the Veh_State, Veh_Cntrls, GUI_Mgr, ENV_Mgr, VDYN_Mgr, IG_Mgr, and Ent_Mgr
- sets non-blocking terminal I/O
- catch ^C and control-break from keyboard
- disable screen savers

If the user has super-user privileges, then the following steps are also executed:

- turn on the fast timer
- Move clock to processor 0
- Run-on Processor 0
- Restrict Processor 1 for the draw process
- Isolate Processor 1 for the draw process
- lock the whole process (data and text) into memory

For error handling, the Sim_Mgr offers the capability of a debug function and an error function. The prototypes for the two functions are as follows:

```
void debug (char *class_name, char *method_name, char
      *debug_msg, ...);
void error (char *class_name, char *method_name, char
      *error_msg, ...);
```

Both functions will print an error message. *error*() will, however, exit the simulation and properly close down all managers. It will exit the program directly through the standard function *exit*(). The debug() method, returns control to the calling object. An object class name and the method name from which *debug*() or *error*() were called may be supplied. These names will precede the error message.

The following is a list of the public methods in the Sim_Mgr object. The names of the methods describe their function:

```
void hide_GUI_Mgr();
void show_GUI_Mgr();
LogicalType get_GUI_Mgr_status();
void initialize_simulation(int vehicle_model, float
      initial_vehicle_X_pos, float initial_vehicle_Y_pos,
      float initial_vehicle_heading, short site, short host,
      short entity);
void run_simulation();
void stop();
```

```
void printOn();
void debug(char *class_name, char *method_name, char
      *debug_msg, ...);
void error(char *class_name, char *method_name, char
      *error_msg, ...);
```

3.2.3 Image Generator Manager (IG_Mgr) Object

For a list of all attributes and methods, please refer to the source code header file IG_Mgr.h++. The primary methods for this object are described below.

The IG Manager constructor initializes Performer, shared memory, the object command pipes, and multi-processing functionality, if available. Because other managers may need to use the shared memory space, especially the command pipe structures, the IG_Mgr should be created before all other managers, except for the Vehicle State and Vehicle Dynamics objects. Pointers to these objects are passed to the IG_Mgr. At the very least, these two objects should not try to use shared memory until their initialization functions (not the same as the constructors) are called. Every object should have the method *initialize*().

The destructor, of course, merely frees the shared memory space and then exits Performer. It is **very** important that the IG_Mgr be deleted last. When the Performer pfExit() function is called, it actually makes a call to the standard function *exit*() which will leave the simulation. Though this result may be desired, it may also be premature if the IG_Mgr is deleted before another object needs to execute its destructor.

The IG_Mgr *initialize*() function performs a great deal of work. It opens the window and initializes all the graphics hardware. It also constructs the default fog, lighting, and earth/sky models. (For more information on these models, please refer to the Performer manuals [SGI92a, SGI92b]). The IG_Mgr loads the flight terrain database for the simulation. This is accomplished via a call to *LoadFlt*(), a function provided by Software Systems, Inc. for IRIS Performer. If Dynamic Terrain is to be used, then the IG instantiates the DTMObj. This object is fully detailed in Robert Buckley's Master's thesis on Bwanavision [Buckley93].

The *update*() method sets the current viewpoint and orientation and then starts scene traversal. The call to *pfSync*() momentarily pauses the program until the next frame boundary. This will keep the simulation running at a specified frame rate, if it is desired. Otherwise, it does not impair the speed of the simulation. The call to *pfFrame*() begins the culling of the scene geometry. Culling will remove any objects or polygons which lie outside the viewing frustum. The objects which are still to be drawn are then sent down the pipe to the draw process which renders the scene onto the channel (window). This entire procedure is described more in depth in the Performer manuals [SGI92a, SGI92b]. The draw process then modifies the current view position based on the vehicle state object. and the visual changes of the vehicle itself. This includes the position and orientation of the vehicle, and the angles at which the bridging mechanism or bulldozer blade has been rotated. *update*() also calls the vehicle dynamics update function.

3.2.4 Vehicle Dynamics Manager (VDYN_Mgr) Object

For a list of all attributes and methods, please refer to the source code header file VDYN_Mgr.h++. The primary methods for this object are described below.

The Vehicle Dynamics Manager operates in two distinct ways. First, it is the managing object for all of the objects that actually simulate the vehicle (i.e., those that control and simulate the vehicle's gross motion and part articulation). It calls these different objects as they are needed based on the current operating state of the vehicle. During dynamically correct simulation, it calls mobility and/or bridging. If only kinematic simulation is requested, it calls the kinematics object (which only simulates gross vehicle motion, no articulations). When switching between dynamics and kinematics, it uses the data stored in the Vehicle State object to insure continuity of motion.

The second task of the Vehicle Dynamics Manager is to simulate the workings of the internal components and systems of the vehicle (i.e., engine, transmission, steering, brakes, and hydraulics). It does this for two reasons. For one, it keeps the two dynamics objects generic by taking advantage of how different types of internal components can run the same vehicular systems (i.e., different engines and transmissions may require very different models but the functionality of track driven vehicles remains the same). This approach also allows the Vehicle Dynamics Manager to evaluate the current operating state of the vehicle. Because it's running those systems and therefore directly interfacing with the dynamics models, it can select when to run either one or both of them. In addition, it allows the mobility button in the user interface (which turns the dynamics on and off) to work through the same interface as the vehicle controls.

The controls interface is a direct command pipe between the Vehicle Dynamics Manager and the GUI Manager. This direct connection is how all of the changes to the controls get transmitted into the simulation of the vehicles engine, transmission, etcetera. Since it also functions two ways, it allows the VDYN_Mgr to tell the GUI manager not to perform certain operations (e.g., preventing the transmission from shifting gears at the incorrect time or preventing the hydraulic clutch for the bridging system from engaging). It also allows for the dynamics input data to be sent along with the commands to simulate the vehicle. For the mobility object, the current speeds of the left and right drive sprockets are sent. The bridging object receives the pressure in the hydraulic cylinders.

The following is a list of the public methods in the VDYN_Mgr object. Essentially, these methods initialize the VDYN_Mgr with vehicle-specific information and then update the dynamics every cycle of the simulation loop:

```
void initialize()
void update (float time_step)
```

3.2.4.1 Mobility Object

For a list of all attributes and methods, please refer to the source code header file Mobility.h++ and Trafficability.h++. The primary methods for this object are described below.

The Mobility object evaluates all the dynamics and trafficability equations for simulating the motion of the vehicle. It receives the current drive rotational speeds of the left and right side drive sprockets when called by the Vehicle Dynamics Manager. Using these and the current dynamic state of the vehicle, it processes the trafficability algorithms to determine what ground forces are transmitted through to the vehicle.

For this implementation, two different Mobility objects were created. One object implemented the WES Mobility model and the second implemented the Bekker/Wong algorithms. The Bekker/Wong version is referred to as the Mobility object and uses the Modified Euler's Method to predict the vehicle's motion. The WES-based mobility algorithms are implemented in the Trafficability object and uses a Runge-Kutta integration method. These two objects are essentially interchangeable at compile time.

Environmental geometry and soil property data are retrieved from the Environment Manager object through the use of the Soil Sample object. The resultant motion of the vehicle is transmitted to the Vehicle State object for use by the rest of the simulation system. These interactions are shown in Figure 16 and are the same whether the Mobility or Trafficability object is used.

The following is a list of the public methods in the Mobility object:

```
void initialize()
void configure()
int simulate (float time_step, float left_track_speed, float
      right_track_speed)
```

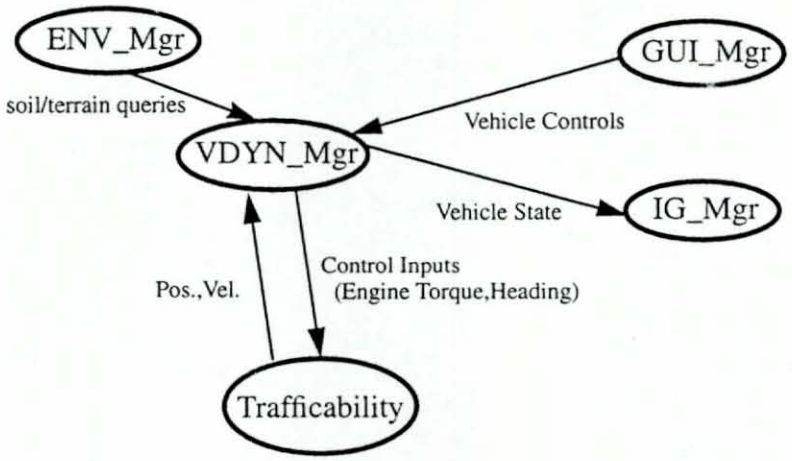


FIGURE 16: Object Interaction for Trafficability (Mobility)

3.2.4.2 Trafficability Object

The Trafficability object has a local state for the vehicle velocity, position, acceleration, and orientation. It has additional states for tracks. It also has methods to implement the track-soil dynamics along with the state and methods of a local soil object. This local Soil object has its own soil structure and methods to manipulate the existing soil properties to

obtain the other soil properties needed to calculate shear force, normal force, etc. The interaction of the Trafficability and Soil objects are shown in

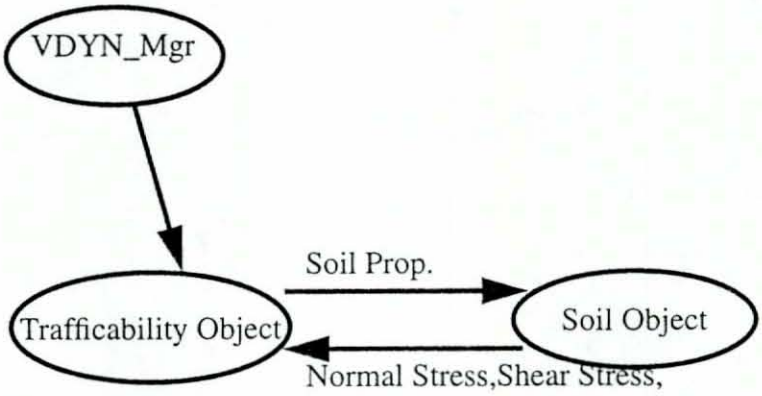


FIGURE 17: Interaction of Trafficability and Soil Objects

Some of the structure for the local states (i.e., attributes) in the Trafficability object are as follows.

- struct Vehicle_Param Curnt_Veh_Param,*Veh_Param;
- struct Vehicle_State Curnt_Veh_State,*Global_Veh_State;
- struct Vehicle_State Tmp_Veh_State, *Veh_State;
- struct Track_Param Curnt_Veh_Track_Param, *Track_Param;
- struct Track_Obj Right_Track,Left_Track, *Right_Wheel, *Left_Wheel;

The following are the private methods in the trafficability object. Descriptions of some of them are self explanatory while others are explained where necessary:

```
void Init_Soil_Prop();
```

- Initializes some of the soil property constants into the local soil object. It is also used to initialize the analysis type, failure criterion. It is useful while running this model separately without GUI, IG support. Currently some of the soil property are obtained by querying the environment manager.

```
void Init_Veh_Param();
```

- Initializes the vehicle specific parameters at start-up viz. Vehicle gross weight, grouser factor, track factor, bogie factor, clearance factor, engine factor, Transmission factor, Vehicle length, width, No of wheels, moment of inertia values, Integration scheme and integration step length. The configuration is read from an ascii text file called "Veh_Config".

```
void Init_Track_Param();
```

- Initializes track specific parameters viz. number of bogie wheels, number of

pads, Track length, Track width, Track area and distance between the tracks. The track parameters are read at start-up from an ascii text file called "Track Param".

```
void Init_Veh_State();
```

- Initializes the vehicle state viz. position, heading at start-up. These values are supplied by the VDYN_Mgr and initialized into the local vehicle state.
- The local vehicle state aids in storing the vehicle state for multi step integrations or while running trafficability at a different frequency than the main program.

```
void Init_Right_Track();
```

- The wheel offsets are currently stored here.

```
float Vehicle_CI();
```

- This method computes the Vehicle Cone index(VCI). This is calculated once when vehicle parameters are initialized.

```
float Coeff_Roll_Resistance(float);
```

- This has to be computed for each track point in contact with soil and from where soil properties are taken. They depend on the soil type, VCI and CI.

```
float Find_Resultant_Velocity();
```

- Finds the resultant vehicle velocity.

```
float Find_Inner_Track_Velocity();
float Find_Outer_Track_Velocity();
float Find_Slip_Velocity(float,float);

float Find_Slip_Displacement(float,float,float);
```

- Finds the shear displacement of the track point under consideration.

```
void Find_MInertia(vector *);
```

- Finds the mass moment of inertia of the vehicle for dynamic equilibrium considerations.

```
void Get_Terrain_Property();
```

- Initializes the local soil class variables for each track point (currently under each wheel) with the soil property returned by the environment manager. Cone index and depth of penetration of the cone penetrometer should be available for this part of the implementation. Since they are not available currently in the database they are calculated using the available soil properties for an average penetration depth.

```
void Pick_Sample_At();
```

- This method computes the current position on the track where soil properties are required, based on the heading and position of the vehicle.

Some of the public methods are as follows:

```
void Run_Trafficability(float []);
```

- Returns the accelerations required by the integrator. This calls the necessary methods to compute the slip, shear displacement and the other Class_Soil methods to obtain the shear and normal stresses. The equations of motion are also implemented here.

```
void RK4(float);
void RK2(float);
void Euler(float);
void Update();
```

- Updates the local vehicle states and computes the heading.

```
void initialize();
```

- Initializes the local vehicle state with the initial vehicle position and heading

```
void write_output();
```

- The new position and heading of the vehicle is updated in the global vehicle structure.

```
void Simulate(float,float,float,float);
```

- The VDYN_Mgr invokes this method by providing with the time step, Left and Right sprocket speeds or the Left and Right shaft torques. The different mobility models could be chosen in the VDYN_Mgr by either calling Class_Trafficability.Simulate or Class_Mobility.Simulate. This method calls Run_Trafficability to compute the accelerations and then integrates with either one of the integrators. It also calls Find_Normal_Vector and updates the local and global vehicle states.

```
void Find_Normal_Vector();
```

Psuedocode which outlines the use of the attributes and methods in the Trafficability object are shown in Figure 18.

3.2.4.3 Soil Object

The local soil object is defined in the class Class_Soil and the other attributes can be seen in the header file Class_Soil.h++. Some of the public methods described are as follows:

```
float Rigidity_Modulus(Class_Soil_Properties
      *Soil_Attributes);
```

- The rigidity modulus of the soil sample is computed using the cone index obtained and based on the void ratio for that soil type.

```
float Apparent_Rigidity_Modulus(Class_Soil_Properties
     *Soil_Attributes);
```

- The Apparent rigidity modulus is the modified rigidity modulus taking into account the surface effects.

```
float Cone_Index(Class_Soil_Properties *Soil_Attributes);
```

- The cone index is recomputed using the apparent rigidity modulus and other soil parameters. These relations are developed by WES [WES 86]. The cone index measured gives a higher estimate than the one recomputed here.

```
float Normal_Stress(float,int,Track_Param
      *Trk_Param,Vehicle_State *Veh_State,
 Vehicle_Param *Veh_Param);
```

- The normal stress is calculated using the contributions due to weight and inertia terms.

```
void Shear_Stress(float, float, float,int,float,Track_Param
      *Trk_Param,
Vehicle_Param *Veh_Param,float [2] ,Class_Soil_Properties
      *Soil_Prop);
```

- The shear stress is calculated using the normal stress , shear deformation, rigidity modulus and other terms.

The pseudocode for the implementation of the above routines is shown in Figure 18.

Initialize

Init_Veh_Param
Init_Veh_State
Init_Track_Param
Init_Right_Track
Init_Left_Track
Init_Soil_Prop

Simulate (Inner shaft speed, Outer shaft speed)

foreach Δt

use RK4,Euler *(integrators) to*

∫ *acceleration* → *(to get Vel. , Pos.)*

Pick_Sample_At
Get_Terrain_Property
Find_MInertia
foreach track
foreach point_on_track
read_soil_Property
Find_Rigidity_Modulus
read_cone_index
Find_Coeff_Roll_Resistance
Find_Slip_Velocity
Find_Slip_Distance
Find_Normal_Stress
Find_Shear_Stress
Find_Transverse_Shear
Find_Tractive_Force
Find_Normal_Force
Find_Acceleration

Find_Normal_Vector *(to obtain new position to pick soil sample)*

Write_Output *(post the new Pos., Orientation)*

FIGURE 18: Pseudocode for Trafficability Dynamics

3.2.4.4 Bridging Object

For a list of all attributes and methods, please refer to the source code header file Bridging.h++. The primary methods for this object are described below.

The Bridging object evaluates all of the dynamics and ground contact equations for simulating the articulation of the bridging system components. It receives the current pressures in the hydraulic systems and uses them and the bridge's current state to dynamically predict the motion of the bridge components. It does this in one of three different modes, one for each of the major operations that the AVLB undergoes when deploying and retrieving bridges (see Figure 5). It also models the detachment and reattachment of the bridge from the launcher.

Environmental geometry and soil property data are retrieved from the Environment Manager object through the use of the Soil Sample object. The resultant motion of the bridge components is transmitted to the Vehicle State object for use by the rest of the simulation system. Attachment and detachment messages are sent directly to the Entity Manager.

The following is a list of the public methods in the Bridging object. The method names describe their function:

```
void initialize()
void configure()
int simulate_overhead (float time_step, float tongue_lever)
int simulate_tongue_n_scissor (float time_step, float
      tongue_lever, float scissor_lever)
void detach_bridge()
int simulate_tongue_n_overhead (float time_step, float
      tongue_lever, float overhead_lever)
void reattach_bridge()
```

3.2.4.5 Kinematics Object

For a list of all attributes and methods, please refer to the source code header file Kinematics.h++. The primary methods for this object are described below.

The Kinematics object is a very simplified version of the Mobility object. It exists so that comparisons can be made between the vehicle's behavior with and without considerations of mobility. It functions with the same interfaces but does not consider Newtonian physics or the properties and slope of the underlying terrain.

Environmental geometry is retrieved from the Environment Manager object through the use of the Soil Sample object. The resultant motion of the vehicle is transmitted to the Vehicle State object for use by the rest of the simulation system.

The following is a list of the public methods in the Kinematics object:

```
void initialize()
void configure()
```

```
int simulate (float time_step, float left_track_speed, float
      right_track_speed)
```

3.2.5 Graphical User Interface Manager (GUI_Mgr) Object

For a list of all attributes and methods, please refer to the source header file GUI_Mgr.h++. The primary methods for this object are described below.

The GUI object constructor initializes the FORMS interface, creating all FORMS objects to be used during the simulation. Keyboard keys are queued for later processing by the IG_Mgr. Only the draw process, that process which opened the graphics window, has the ability to read the queue. Although the GUI has indeed opened a window, it may not be able to access any keyboard events in the IG window. Therefore, we leave all keyboard processing to the IG.

The GUI *initialize*() function is currently empty because all initialization is accomplished in the constructor.

As expected, the destructor hides the visual interface and frees all the memory consumed by the FORMS objects. This function closes the control panel window.

3.2.6 Environment Manager (ENV_Mgr) Object

For a list of all attributes and methods, please refer to the source code header file ENV_Mgr.h++. The primary methods for this object are described below.

The ENV_Mgr performs terrain following by intersection with the Performer geometry scene. This is possible whether or not the scene geometry was loaded from a Flight file or generated by the dynamic terrain process. In either case, the ENV_Mgr is oblivious to where the polygonal data originates. The terrain itself, as explained in a preceding section, is created or loaded by the IG_Mgr. More about the dynamic terrain aspect of the TVS is explained in the section on the IG_Mgr.

The methods for querying the terrain are as follows:

```
long get_terrain_elev_n_surf_norm (Class_Soil_Sample
      *soil_sample, int n);
void get_soil_properties (Class_Soil_Sample *soil_sample,
      int n);
long get_all_terra_attributes (Class_Soil_Sample
      *soil_sample, int n);
```

All three routines accept an array of soil samples from the calling function and an integer describing the number of objects in the array. This number cannot be larger than the number passed to the ENV_Mgr *initialize*() method, which allocates the maximum number of query points. *get_terrain_elev_n_surf_norm*() returns the separate elevation and normal values for the soil sample points. Vectors are supplied in the samples describing their direction of intersection with the terrain. In the TVS, these vectors are equivalent for a given query across the soil samples, but this is not required by the

method; they may be in random directions and at random locations.

Calling *get_terrain_elev_n_surf_norm*() will cause the ENV_Mgr to perform intersection testing of the given segments and the terrain geometry loaded on the Performer scene. The resulting intersection points are then stored back into the soil samples corresponding to the correct point/segment. The normals of the terrain at those intersection points are also saved in the soil samples. The geometry in this scene was stored there by the IG_Mgr when either loading the terrain Flight file or creating the DTMObj. The DTMObj dynamically implements polygons on the geometry scene which, of course, the ENV_Mgr can access.

get_soil_properties() will fill in the values for the soil properties. These variables allow the realistic mobility model of the vehicle dynamics manager to operate correctly. For more information on what these properties are and how they affect the mobility of the vehicle, please refer to Section 2.6 and Section 2.7.

A method to easily implement the above capabilities has been supplied. It is *get_all_terra_attributes*(). As you might expect, *get_all_terra_attributes*() merely makes calls to the two functions described above.

3.2.7 Entity Manager Object

To visualize other entities, TVS uses portions of BwanaVision [Buckley93]. BwanaVision does not communicate with the Entity Services program. As explained in the thesis, BV interfaces with another program called DIP (DIS Interface Process). However, the TVS needs to use the Entity Services for easy updates to the PDU structures. It was a relatively simple task to design a new PDU type such as the Attach PDU and implement it in this project. However, a new BV entity manager object, *ESERV_EntityManager*, was created to communicate between Ent_Mgr and ESERV. This new entity manager is similar to the BV object *NTSC_EntityManager*, which communicates through shared memory to DIP. The *ESERV_EntityManager* now communicates to ESERV and also allows for transmitting PDUs of any type. These relationships are shown in Figure 12.

The TVS Entity Manager needs to update its entity state correctly and therefore provides the following methods in its public definition:

```
 void initialize(short site, short host, short entity);
 void update();
void set_entity_xyz(float x, float y, float z);
```

- set the current world coordinates of the vehicle

```
void set_entity_hpr(float h, float p, float r);
```

- set the current orientation of the vehicle

```
void set_entity_vel(float x, float y, float z);
```

- set the velocity (for dead reckoning)

```
void set_entity_acc(float x, float y, float z);
```

- set the acceleration (for dead reckoning)

```
void set_entity_ang_vel(float h, float p, float r);
```

- set the angular velocity (for dead reckoning)

```
void set_art_part(int part, float value, float value2);
```

- set the values for an articulated part

```
void set_entity_destroyed(void);
```

- when the entity is destroyed, then this flag must be set

```
void detach_entity(void);
```

- used to detach the articulated parts from the entity, such as the bridging mechanism on the AVLB, sending out an Attach PDU

```
void attach_entity(EntityID3 id);
```

- re-attaches the articulated parts onto the entity, sending out another Attach PDU

```
void set_detach_id(short site, short host, short entity);
```

- set the ID of the new entity to be created by the detachment of the current entity's articulated parts

```
void set_detach_position(float x, float y, float z);
```

- set the world position of the detached entity

```
void set_detach_orientation(float h, float p, float r);
```

- set the orientation of the detached entity

```
EntityMsg *ExtrapolateBridges(int *n);
```

- scan the net for all AVLB bridges which have been detached; this is used by the Vehicle Dynamics Manager when the host AVLB (for this simulation) attempts to attach to a bridge

The Entity Manager, through the ESERV_EntityManager in BV, reads the entity state information from the DIS network. This information must be translated into visual models to be displayed in the IG Manager. Moreover, detonations should be displayed on screen. Therefore, these polygons (for explosions and entities) must be placed on the visual geometry scene. This is the same scene which the ENV_Mgr uses for terrain intersection testing, as explained in an earlier section. Flags on this geometry ensure that they are not

used for intersection testing. If a bridge entity is encountered, however, then the bridge geometry is flagged to take part in intersection testing so that the ownship may drive over the bridge. Thus, the Ent_Mgr generates the polygons for the IG_Mgr. Specifically, a Flight model file is loaded or cloned for every new entity. To understand cloning better, please refer to the Performer manuals under the function name *pfClone*().

Detach and Attach messages are sent through the Ent_Mgr by the VDYN_Mgr; specifically by a local instantiation of the Bridging object. The Bridging object must supply the current world position and orientation of the bridge when detaching it. It must also make a call to *ExtrapolateBridges*() when determining if a bridge entity on the network can be re-attached. The mobility model sends the vehicle's ownship entity information such as the velocity, acceleration, and angular velocity to the Ent_Mgr for its continuous updates on the DIS network.

3.2.8 Shared Memory

Performer provides an easy way to allocate and access shared memory. Although the TVS does not currently support multi-processing, it is useful to view each manager as a different process. In this respect, the IG_Mgr and GUI_Mgr need a quicker and more reliable way to communicate than the command pipe processor for latency dependent variables (which affect important objects like the viewpoint). We can declare some variables in shared memory (such as the command pipe file descriptions themselves) and then every object will be able to access them. Performer makes this task as easy as declaring any other variables. The shared memory space merely needs to be allocated between pfInit and pfConfig, two Performer functions called in the IG_Mgr. Performer then handles all shared memory constructs, such as data locking and semaphore control. The following C construct describes the TVS shared memory structure:

```
typedef struct
{
short viewScreen;
short viewChanged;
long IGWindow;
int reality_graphics;
int IGcmdpipe[2];
int GUIcmdpipe[2];
int VDYNcmdpipe[2];
int ENVcmdpipe[2];
pfScene *scene;
} SharedData;
```

viewScreen contains the current viewing direction of the IG. The possible choices are front view, rear view, stealth view, left view, and right view. *viewChanged* will be true when the view angles or position needs to be modified. This process could require a fair amount of calculation, so it is only performed when necessary. In practice, however, it does not seem to slow down the system very much, if at all. *IGWindow* is the identification value for the window the IG_Mgr has opened up. The GUI_Mgr needs to reference the ID of the IG window after modifying the user input windows. If the system is running on a machine

with Reality Engine graphics, then the variable *reality_graphics* will be set to true by the IG_Mgr. scene is the root pointer to all visual models which may be displayed by the IG. This includes any terrain geometry, vehicle geometry by the ownship, other entity geometry, and any cultural and feature data. The other variables: IGcmdpipe, GUIcmdpipe, VDYNcmdpipe, and ENVcmdpipe, are the command pipes for the respective simulation managers.

3.3 The Bridge DTR

3.3.1 Software Overview

Once a bridge is deployed by a simulated vehicle such as the AVLB, the Terrain Manager detects this activity via the Attach/Detach PDU and activates the Bridge DTR. The Bridge DTR then serves as the simulator for the deployed bridge which now becomes a separate entity on the DIS network.

The bridge simulation, run by the Bridge DTR, then determines the state of the deployed bridge. This state is based on effects such as the type of soil the bridge is deployed on, the orientation of the bridge once it is deployed, and the effect of vehicular traffic moving over the bridge. After initial deployment, the dynamics model for the bridge determines any movement of the bridge due to the terrain slope and the type of soil before the bridge finally comes to rest. As described in Section 2.5, this model uses many of the same algorithms within the vehicle mobility model to determine slip and resistive forces due to the terrain.

Once the bridge is deployed, the effects of vehicular traffic are taken into consideration. Through use of Entity Services [VSL94], entities are monitored until they come within close proximity of the bridge. As they approach and drive over the bridge, the effects of their weight on the bridge can be calculated to determine if the bridge collapses due to the total load.

The Bridge DTR is sufficiently flexible such that it can simulate multiple bridges simultaneously. Thus, the Bridge DTR can manage multiple deployed bridges and, for each bridge, monitor the vehicular traffic and the effects of that traffic on a bridge.

3.3.2 Detailed Design of the Bridge DTR

As shown in Figure 19, the Bridge DTR is a separate process that consists of several objects or modules. To maintain a list of the bridges managed by the Bridge DTR, the BridgeElement object represents a bridge within the list. The list, referred to as the Bridges object, is instantiated from a doubly linked list class (DLLIST) that was written for the Dynamic Terrain Testbed software. Each Bridges list is populated when the Bridge DTR process receives entity state information about ground vehicles deploying or retrieving bridges.

For each bridge, a list of vehicles within close proximity must be maintained. These vehicles, which apply forces to the bridge through their motion and mass, are represented

by the BridgeWeightElement objects. Each of these BridgeWeightElement objects represent a single vehicle within the list of all vehicles stored for each bridge. This list of vehicles, called the BridgeWeights object, is also instantiated from the DLLIST class of doubly linked lists. Each BridgeWeights list is populated when the Bridge DTR process receives entity state information about ground vehicles approaching a bridge.

In addition to monitoring the states of ground vehicles, the Bridge DTR determines the state of bridges under it's control by associating a DeployedBridge object with each BridgeElement. This object calculates the motion of the bridge due to its placement on the terrain and the soil type and condition. This data is obtained by the Environment Manager (ENV_Mgr) object. By accessing the list of BridgeWeightElement objects, the DeployedBridge object also computes the forces exerted on the bridge by vehicles on or near the bridge. These objects used by the Bridge DTR are all written in C++ and are described in more detail below.

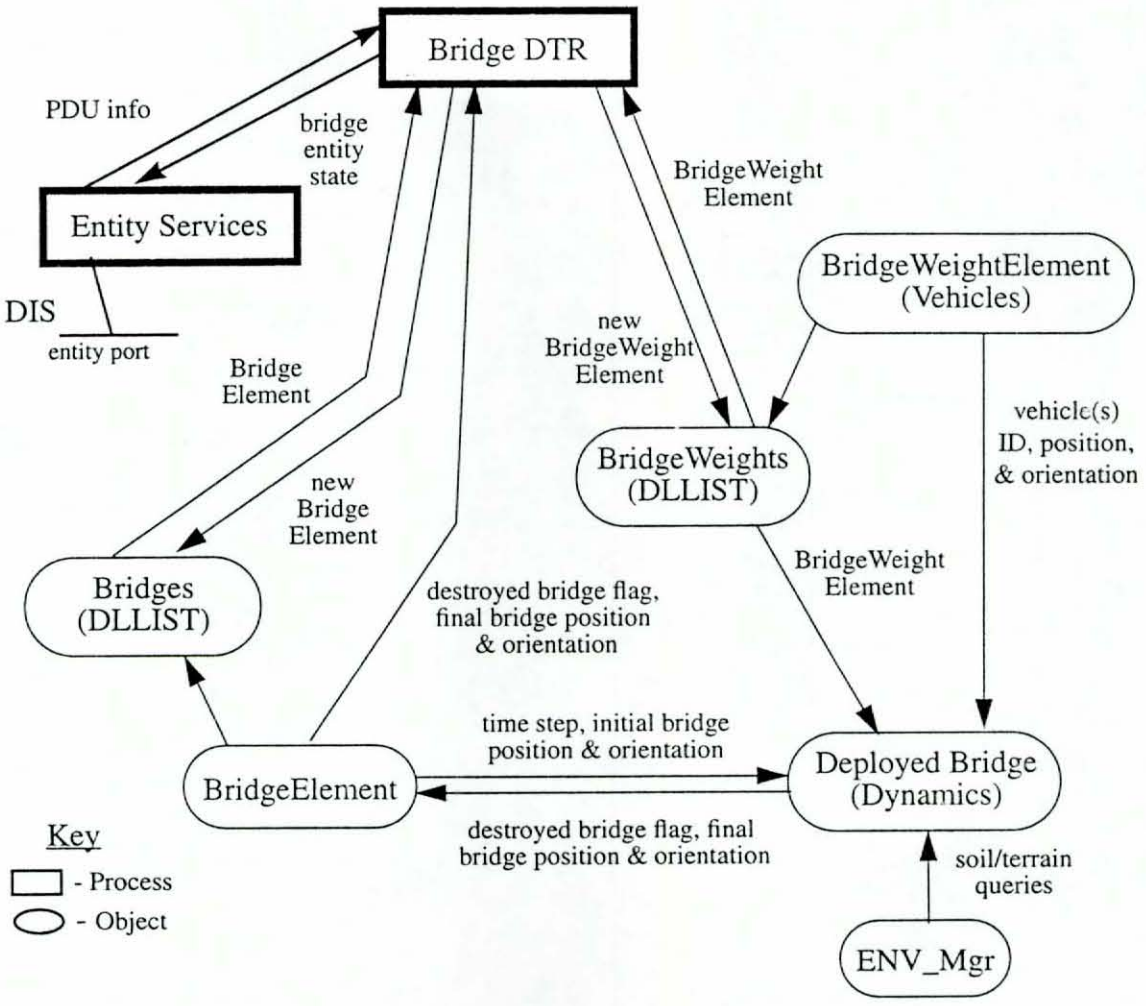


FIGURE 19: Data Flow and Object Relationships of the Bridge DTR

3.3.2.1 Bridge DTR Process

As described previously, the Bridge DTR is responsible for maintaining the state of one or more deployed bridges as well as determining the effects of vehicles driving over these bridges. For this purpose, the Bridge DTR serves as the "main" process and generates all objects used in the bridge simulations. Below is psuedocode for the Bridge DTR. Remember that a BridgeElement refers to a bridge being managed by the Bridge DTR process and a BridgeWeight is the weight applied to a bridge by a vehicle entity.

```
establish connection to Entity Services process
instantiate Environmental Manager object and initialize to process 12 terrain
attribute property requests at a time (for bridge placement and dynamics)
LOOP WHILE end_program flag is FALSE
    LOOP WHILE awaiting ATTACH/DETACH PDU from Entity Services
        create temporary Bridge Element object when PDU recieved
        IF DETACH PDU
            IF bridge already in Bridges list
                update bridge state
            ELSE
                add bridge element to Bridges list
            END IF
        ELSE IF ATTACH PDU
            find bridge (i.e., bridge element) in Bridges list and remove
        END IF
    END WHILE LOOP
    LOOP WHILE awaiting ENTITY STATE PDU from Entity Services
        FOR each bridge (BridgeElement) in Bridges list
            compute distance between entity and bridge (i.e., BridgeElement)
            FOR each entity in the BridgeWeights list
                IF current entity already in BridgeWeights list
                    IF entity within BridgeElement bounding sphere
                        update BridgeWeightElement with entity state
                    ELSE
                        remove entity from BridgeWeights list
                    END IF
                ELSE IF entity within BridgeElement bounding sphere
                    add entity (BridgeWeightsElement) to BridgeWeights list
                END IF
            END FOR LOOP
        END FOR LOOP
    END WHILE LOOP
    FOR each bridge (BridgeElement) in Bridges list
        simulate bridge
```

- compute affects of vehicles driving over and underlying terrain
- copy bridge state into ENTITY STATE PDU
- send bridge ENTITY STATE PDU to Entity Services for transmission
- END FOR LOOP
- clean up lists (Bridges and BridgeWeights)
- END WHILE LOOP

3.3.2.2 Bridges and BridgeWeight List Objects (DLLIST, DLLIST_Iterator, and Element Class)

The Bridges and BridgeWeight list objects are instantiated from the doubly linked list (DLLIST) class. The list has a current element which can be used for primary traversal and updates to the list. Methods such as addElemBeforeCurrent and addElemAfterCurrent can be used to add list entries before or after the current element of the list.

The DLLIST class contains the following attributes:

- friend class DLList_Iterator;
- typedef struct NodeStruct Node;
- struct NodeStruct {Element *elem; Node *prev, *next; };
- Node *Head, *Tail, *Current, *Iterator, *Temp, *AddAt;
- Element *TempElem;
- int NumberOfElements, ElementNumber;

The following is a list of the public methods in the DLLIST class. The names of the methods define their function:

```
void clearList();
void moveToFront();
void moveToEnd();
void addElemToFront(Element*);
void addElemBeforeCurrent(Element*);
void addElemAfterCurrent(Element*);
void addElemToEnd(Element*);
int  prevElem(); // returns atFronOfList();
int  nextElem(); // returns atEndOfList
Element* firstElem();
Element* currentElem();
Element* lastElem();
const Element* viewFirstElem();
const Element* viewCurrentElem();
const Element* viewLastElem();
void removeFront();
void removeEnd();
void removeCurrent();
int  atFrontOfList();
int  atEndOfList();
int numberOfElements() {return NumberOfElements;}
```

```
int  atElementNumber() {return ElementNumber;}
```

To search all elements of a list for a particular element, the DLLIST_Iterator class was created. The DLList_Iterator (Doubly Linked List Iterator) is useful for quick scans of elements in the list or in the case where a particular element or item must be accessed for each element. The iterator allows for traversal of the list with the operator() and the view() methods returning NULL when they reach the end of the list. Note that concurrency problems can arise from unprotected changes to the list in the middle of an iteration pass.

The DLLIST_Iterator class contains the following attributes:

- int PositionInList;
- DLList *List;
- Node *Current;

The following is a list of the public methods in the DLLIST_Iterator class. The method names describe their function:

```
DLList_Iterator(DLList&);
DLList_Iterator(DLList_Iterator&); // sync on another
      iterator
void init();
void moveToFront();
void moveToEnd();
int  atFront();
 int  atEnd();
 Element* operator() ();
const Element* view();
void operator++();
void operator--();
void setListCurrentToCurrentIter();
void removeCurrentIter();
void updateCurrentIter(Element*);
void moveElement(DLList_Iterator& from, int before); // to
      where this iterator is
int  positionInList();
```

Each Bridge in the Bridges list and each vehicle or entity in the BridgeWeight list is derived from the parent Element class. This class allows any class that inherits from the Element class to be put into the list. The main defining feature of heirs to Element is the implementation of the new_object method. Their are no attributes defined for this class. However the public methods for this class are:

```
virtual Element* new_object();
```

- create a new Element

```
virtual int isA();
```

- determine if the Element is the correct type

```
virtual void printOn();
```

- output debugging information

3.3.2.3 BridgeElement Object

This object class, derived from the Element class, represents a single bridge being maintained by the Bridge DTR through the Bridges list. The attributes, shown below, consist of entity state information for the bridge as well as links to the Environment Manager, the Deployed_Bridge object, and a list of vehicles near or on this bridge (i.e., BridgeWeights).

- Class_ENV_Mgr* env_manager;
- Class_Deployed_Bridge* deployed_bridge;
- int destroyed;
- int site, host, id;
- double x, y, z;
- float heading, pitch, roll;
- DLList BridgeWeights;
- BridgeWeightElem* WeightAccessor;

The public methods for the BridgeElement class are shown below:

```
Element* new_object() {return ::new BridgeElem(*this);}
```

- create a new Bridge

```
void simulate();
```

- compute the dynamics of the bridge to determine its state

```
void operator= (const AttachMsg&);
```

- set new Bridge to current bridge

```
int operator== (const BridgeElem&);
```

- determine if two Bridges are equivalent

```
void copyIntoEntityMsg(EntityMsg&);
```

- generate an Entity State PDU for the Bridge state

```
void printOn();
```

- output debugging information

3.3.2.4 DeployedBridge Object

The DeployedBridge Object contains the algorithms to simulate the dynamics of the bridge. Each BridgeElement has associated with it a single DeployedBridge object. With this approach, the simulation control for the bridge (BridgeElement) is separate from the bridge dynamics (DeployedBridge). The attributes and methods used for the DeployedBridge object are similar to the VDYN_Mgr object in the TVS. The attributes of the DeployedBridge object are variables used in the dynamics calculations and are too numerous to list here. Instead, these variables are documented in the source code. The public methods used to initialize the constants for the dynamics calculations and to run the simulation are shown below:

```
void initialize (float, float, float, float, float, float);
int simulate (float pass_time_step);
```

The simulate method is of primary interest. As shown in Figure 19, this method is called by the associated BridgeElement object which passes the time step and the position and orientation of the bridge since the last simulation run. The method returns the final bridge position and orientation of the bridge based on the soil the bridge rests on and the vehicles driving over the bridge. The vehicles are accessed by the DeployedBridge object through the BridgeWeights list. For each vehicle in the list, a BridgeWeightsElement object is queried which provides the vehicle ID, position, and orientation. The vehicle ID is used to ascertain the type and weight of the vehicle. If the total weight of the vehicles on the bridge is greater than the maximum load of the bridge, a destroyed bridge flag is also returned by the simulate method of the DeployedBridge object.

3.3.2.5 BridgeWeight Object

As described previously, each BridgeWeight object represents a vehicle or other entity applying a force to a bridge. The attributes of this object are sufficient to identify the vehicle and to compute the force applied by the vehicle to the bridge. The attributes are:

- int active; // used for garbage collection
- int site, host, id;
- double x, y, z;
- float heading, pitch, roll;

The public methods of this object are as follows:

```
Element* new_object(){return ::new
      BridgeWeightElem(*this);};
```

- create new bridge load

```
void operator= (EntityMsg&);
```

- determine if bridge loads are equivalent

```
int operator== (EntityMsg&);
```

- set current bridge load equal to new bridge load

3.3.2.6 Environment Manager (ENV_Mgr) Object

The ENV_Mgr object provides terrain surface and soil property data to the DeployedBridge objects as shown in Figure 19. This is the same object as described in Section 3.2.10. It was found that the ENV_Mgr developed for the TVS also served well for the Bridge DTR.

3.3.2.7 Entity Services Process

This process is the same Entity Services used through the design of the Dynamic Terrain Testbed and described in Section 3.1.3.

3.4 Using the Software

3.4.1 Requirements

3.4.1.1 System Requirements

The TVS requires Performer 1.2.72a or later. A modification of the make file allows the user to switch compilation between IRIS Performer version 1.0 and 1.2. However, upon the introduction of the dynamic terrain implementation, this feature has been aborted. The dynamic terrain code is very large and it would require more effort than allowed to convert the code to run in version 1.0.

The Elan graphics system is the minimum required for this simulator. A Reality Engine system is preferred, but cost can also be a factor to consider. The TVS was developed on a series of Indigo Elan systems with the MIPS R4000 (50Mhz, doubled) processors. Even this powerful system could run a little slow (5Hz or less), depending on ethernet traffic.

3.4.1.2 Input Data Requirements

The vehicle visual model is imported from a Software Systems MultiGen Flight v13.0 data file. In fact, all of the entity visual models are Flight files, though these files may be version 11.0 or later. The terrain file is also a MultiGen Flight file, though it will not be loaded if the system is running dynamic terrain. The dynamic terrain data is described in the configuration file *dtdb.cfg*. The data itself is stored in an elevation file, a two dimensional grid of elevation (Z-value) points. Required soil properties are described in Section 2.6 and Section 2.7.

The only two major input data sets are required for the Bridge DTR. The first is all the environmental data provided by the local Environment Manager. This includes terrain elevations, normal vectors, and soil properties. These are exactly the same pieces of information used by the Tracked Vehicle Simulator itself. Indeed all the same algorithms are used to predict the bridge's behavior. The second set is the kinematic state of all the

close proximity vehicles. This includes the coordinates of the vehicle, it's orientation, and any velocity and acceleration data that is available. This information is contained in the dynamically linked list that is passed by the same part of the system that the bridge dynamics are called from. The position and orientation of the bridge is directly accessed by the parent system from the dynamics objects.

3.4.1.3 Starting and Running the TVS

1. Start Entity Services

   Start entity services by typing *testService* in the entity services directory. The current entity services directory is /home/projects/dt/altman/eserv1/test4. This needs to be run in the proper directory because it uses some configuration files which are located there.

2. Start the bridge DTR

   Start the bridge DTR by typing *bridge* in the bridge DTR directory. The current bridge DTR directory is /home/projects/dt/smith/bridge. This program needs to load in the bridge MultiGen Flight file for its own internal calculations. Please refer to the section on the specifics of the bridge DTR and dynamics for more information.

3. Start the TVS

   Start the Tracked Vehicle Simulator by typing in *TVS* in the TVS directory which is currently /home/projects/hab/release/dttvs/v0.6. This program should be run in the correct directory because there are many files to which it needs access. However, if these files are defined elsewhere, then the TVS may be run elsewhere. Furthermore, many of the model files (for other entities on the net) are stored in separate directories. Therefore, an environment variable called PFPATH should be defined, otherwise errors may occur. Normally, the variable is set upon login in the .login file. The following command, if necessary, should precede starting of the TVS:

```
setenv PFPATH .:/usr/src/Performer/data:/home/projects/dt/
      buckley/flt/Aircraft:/home/projects/dt/buckley/flt/
      Landcraft:/home/projects/dt/buckley/flt/Misc:/home/
      projects/dt/buckley/flt/Seacraft:/home/projects/dt/
      buckley/flt/Spacecraft:/home/projects/dt/buckley/flt/
      Terrain:/home/projects/dt/buckley/flt/Buildings:/home/
      projects/dt/buckley/flt/Textures:/home/projects/dt/
      buckley/flt/Simnet:/home/projects/dt/buckley/flt/hud:/
      usr/mg/Flt/Txt
```

   With this setup, the user should be able to load any of the available entity models, including any texture maps. After running TVS with no command line parameters, the following message will appear:

```
Usage: TVS [options] <model choice>

Options:
-debug = turn debug print statements on (not needed with -D)
```

```
-Txxxx = use terrain filename xxxx (default: MDW.flt)
-Vxxxx = use vehicle filename xxxx (default: <MODEL>.flt)
Warning: Intersection points may not be aligned.
-p x y h = start vehicle at x,y with heading h
-nogui = do not display the GUI (F12 toggles)
-Dxxxx = log debug statements to file xxxx
```

The *-debug* option will print any debug information to the window which the TVS was started from using the standard file definition stderr. the *-D* option will perform the same function except it gives you the opportunity to define the file name. the *-T* option will use the given file name as the terrain file to load (with the non-DT option). The *-V* option allows the user to specify the vehicle file name. This is handy if you have created another higher or lower resolution vehicle model. The *-p* option allows the user to specify the starting position and heading on the command line. The system will automatically determine the Z position, pitch and roll depending on the terrain. The heading is in degrees, from -360.0 to +360.0. All three values are floating point. The last option, *-nogui*, allows the user to start the system up without a user interface. This provides a convenient stealth mode system.

4.0 Data Collection and Testing

Testing of the vehicle dynamics, mobility, and bridging models was severely hampered by lack of available data. For instance, WES does not have a database of cone indices for locations such as Ft. Hunter Liggett, Ft. Hood, or the National Training Center. Since mobility was never in question at these locations, no data collection effort was previously required. However, to simulate terrain properties this data will be required in the future. An additional problem lies in the lack of available vehicle data. Numerous requests were sent through TACOM for engineering data on the AVLB. Some of the requested data arrived nine to twelve months after the initial request. This leads to two possible conclusions: (1) the vehicle data required by the models in Section 2.0 are not currently provided by the manufacturers for some types of vehicles, or (2) the data is not available in a centralized location. This lack of data for terrain and vehicles will make future verification and validation efforts of simulation models extremely difficult.

As specified in Section 2.0, the vehicle models were matched (within ±10%) with the performance specifications in the operations and technical manuals of the AVLB [USARMY86, USARMY91]. However, with insufficient vehicle data, this matching was done through trial-and-error adjustments of vehicle parameters.

5.0 Conclusions

From the development of the Tracked Vehicle Simulator and the associated models for mobility and bridging, a great deal has been learned. These lessons and the suggestions for future work are described in the following sections.

5.1 Lessons Learned

In performing this mobility research, the following lessons were learned:

1. Available environmental models or environmental effects models may not be easily integrated into real-time simulation systems.

   Models previously developed for the Department of Defense to model environmental phenomena, which were not originally considered for use in real-time simulation, may prove difficult to integrate into these types of simulations. For instance, the WES mobility model predicts the maximum speed a vehicle can achieve over a particular homogenous area of terrain. With these assumptions, the model has eliminated a need for vehicle dynamics or a method to model transitions across different terrain types. These assumptions are fine for the original application of the WES mobility models. However, vehicle dynamics and transitions across terrain types are required in a real-time simulation of a ground vehicle. Therefore, significant modifications were required to the WES model delivered to IST in order to incorporate these necessary components of the vehicle simulation.

   Other models may make similar assumptions which eliminate necessary components for real-time simulation. These models may also require modifications that will, in turn, require a significant verification and validation effort.

2. Environmental data required for high fidelity environmental models or environmental effects models may not be readily available.

   As describe in Section 4.0, testing was hindered by lack of available soil property data. The simulation community, with the assistance of agencies such as the Defense Mapping Agency, has been able to accumulate extensive terrain surface data. However, with the incorporation of models such as for ground vehicle mobility, data on the underlying soil properties are now also required. However, there is no known data collection effort for these types of properties. This deficit may also impede future validation efforts of these environmental models.

3. Vehicle data for high fidelity vehicle dynamics models may not be available for some vehicles.

   Increasing fidelity is being sought for simulated vehicles as well as for simulated environments. However, for currently existing ground vehicles,

like the AVLB, data may not be available to supply the vehicle dynamics models used in these high fidelity simulations. As stated in Section 4.0, this may be due to the fact that such data is not required, and therefore, not provided by the manufacturers, or the data is not provided in a central location. This serves as yet another obstacle to validation.

5.2 Recommendations for Future Work

This work could be improved through a number of modifications or future research. For instance, the following modifications could improve the flexibility and fidelity of the tracked vehicle simulator:

1. Modify vehicle dynamics to include motion of articulated parts while the vehicle is moving.

   As suggested in Section 2.2.3 and Section 2.4.3, the vehicle dynamics could be made more generic by allowing changing inertial properties generated through the motion of articulated parts. This would allow for the simulation of a wider variety of ground vehicles.

2. Include high frequency effects on the vehicle due to small obstacles or bumps in the terrain.

   As described in Section 2.2.3, high frequency effects on the vehicle are ignored. However, as terrain database resolutions increase along with higher vehicle fidelity requirements, these effects should be considered.

3. Permit the vehicle to alter the terrain through plowing and formation of vehicle tracks.

   Currently in the Tracked Vehicle Simulator, the vehicle is only affected by the terrain through the mobility model. Modeling of other types of vehicles will require the vehicle to also affect the terrain. For instance, the digging of a bulldozer will change the terrain surface and underlying soil properties. This capability is available in other simulations with the Dynamic Terrain Testbed and should be integrated into the TVS. This should include accurate formation of vehicle tracks which, in turn, will also affect the vehicle mobility based on the sinkage depth. This need has been previously expressed in Section 2.6.4 and Section 2.7.2.

4. Develop a hierarchy of models to simulate different vehicles at different fidelities.

   As stated in Section 2.3, simulation of internal vehicle systems can be complex due to the wide variety of these systems based on the type of vehicle. It is possible that classes of objects can be developed to represent these systems at various fidelity levels and configurations to model existing vehicles as well as concept vehicles under test.

Areas of research that could be pursued to add to the capabilities of vehicle simulations are further enhancements to vehicle model fidelity and physical modeling of objects that can move, but do not have their own propulsion systems.

For instance, vehicles should be modeled as true three dimensional objects that can roll when tipped past their equilibrium point or collide with and physically react to obstacles and other vehicles (see Section 2.2.3). Such behaviors would enhance training by allowing trainees to determine how a vehicle will react to these conditions. For test and evaluation purposes, a more accurate physical model of the vehicle will allow determination of when the vehicle will tip based on the type and grade of terrain, or how it will react to various types of impacts with obstacles or munitions. This requires research and development of improved contact models and flexible object surface models.

Improved contact models would consider contact of the vehicle with the terrain or other vehicles or objects at any point of the vehicle surface. Current contact, or terrain following models, only consider the normal contact points of the vehicle with the terrain (i.e., the vehicle tracks).To model the behaviors listed above, more sophisticated models will need to be developed.

For test and evaluation purposes, flexible structural analysis models will be required to predict the structural damage to a concept vehicle due to various types of collisions.

6.0 References

[Baladi78] Baladi, G.Y., Rohani, B. Proceedings of the Sixth International Conference of the International Society for Terrain Vehicle Systems. Vol.I. 1978.

[BBN88] BBN Systems and Technologies Corporation. SIMNET M1 Abrams Main Battle Tank Simulation - Software Description and Documentation (Revision 1). Report No. 6323. August 1988.

[Bekker56] Bekker, M.G. Theory of Land Locomotion: The Mechanics of Vehicle Mobility. University of Michigan Press. 1956.

[Bekker60] Beker, M.G. "Off-The-Road Locomotion." Research and Development in Terramechanics. 1960.

[Bekker69] Bekker, M.G. Introduction to Terrain-Vehicle Systems. University of Michigan Press. 1969.

[Buckley93] Buckley, Robert Lee, Jr. BwanaVision: A Software Image Generation System Supporting Dynamic Terrain. Masters of Science, Computer Science Project Report. Dept.of Computer Science, University of Central Florida. December 9, 1993.

[DIS94] DIS Interface Subgroup. Software Requirements Specification for the DOD INTERLYNX Network Inteface (Strawman). Facilitated by the Naval Air Warfare Center Training Systems Division. Orlando, FL. January 1994.

[DTIC91] Defense Technical Information Center, Defense Logistics Agency. Technical Report Summaries: AVLB, HAB. July 8, 1991

[IST93a] Institute for Simulation and Training. Distributed Interactive Simulation Operational Concept v2.3. IST-TR-93-25. September 1993.

[IST93b] Institute for Simulation and Training. IEEE Draft Standard Version 2.0.3 Standard for Information Technology - Protocols for DIS Applications. IST-CR-93-15. May 1993.

[Karafiath78] Karafiath, L.L.,Nowatzki, E.A., Soil Mechanics for Off-Road Vehicle Engineering, Trans Tech Publications. 1978

[Kondner63] Kondner, R.L., Hyperbolic Stress-Strain Response: Cohesive Soils. Journal, Soil Mechanics and Foundation Division ASCE. Vol. 89. 1963

[Lisle94] Lisle, Curtis, Altman, Marty, Sartor, Michelle, and Kilby, Mark. "Architectures for Dynamic Terrain and Dynamic Environments in Distributed Interactive Simulation", Summary Report: The Tenth Workshop on Standards for the Interoperability of Defense Simulations. Volume II - Minutes from the Working Sessions. Orlando, FL. March 1994.

[Lucas86] Lucas, G.G., Road Vehicle Performance. Vol. 7. Gordon and Breach Science Publishers. 1986

[Mc Ray64] Mc Ray, J. L. "The Terrain Vehicle Programmes of the US Army Engineer Waterways Experiment Station". Journal of Terramechanics. Vol. 1. 1964

[Minkler] Minkler, G. and Minkler, J. Aerospace Coordinate Systems and Transformations. Magellan Book Co. Baltimore.

[Nakmura91] Nakamura, Shoichiro. Applied Numerical Methods with Software. Prentice Hall. 1991.

[NASA76] NASA. "Quaternions for Control of the Space Shuttle." Lyndon B. Johnson Space Center. Houston, TX. JSC Internal Note JSC-11150. April 1976.

[Overmars91] Overmars, Mark H. FORMS Library: A Graphical User Interface Toolkit for Silicon Graphics Workstations. Version 2.0. Dept. of Computer Science, Utrecht University, Netherlands. Dec. 1991.

[Press91] Press, William H, Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. Numerical Recipes: The Art of Scientific Computing. Cambridge University Press. 1991.

[SGI92a] Silicon Graphics. IRIS Performer Programming Guide. P/N 007-1680-010. 1992.

[SGI92b] Silicon Graphics, Inc. IRIS Performer Man Pages. P/N 007-1681-010. 1992.

[Shabana89] Shabana, Ahmed A. Dynamics of Multibody Systems. John Wiley and Sons. 1989.

[USARMY85] Headquarters, Dept. of the Army. Mobility. Field Manual No. 5-101. January 1985.

[USARMY86] Headquarters, Dept. of the Army. Operator's Manual - Launcher and M60A1 Tank Chassis, Transporting: For Bridge, Armored-Vehicle-Launched, Scissoring Type, Class 60. TM 5-5420-202-10 C2. November 1986.

[USARMY91] Headquarters, Dept. of the Army. Technical Manual - Operator's, Unit, Direct Support, and General Support Maintenance - Bridge, Armored-Vehicle-Launched, Scissoring Type: Aluminum; 60 Foot Span for M48A5 and M60 Launcher (All Makes and Models). TM 5-5420-203-14. June 1991.

[VSL94a] Visual Systems Lab. Terrain Database Services in the Dynamic Terrain Testbed. VSL Internal Report. To be released in Spring 1994.

[VSL94b] Visual Systems Lab. Entity Services in the Dynamic Terrain Testbed. VSL Internal Report. To be released in Spring 1994.

[WES86] Waterways Experiment Station, Corps of Engineers, Dept. of the Army. Steerability Analysis of Tracked Vehicles: Theory and Users Guide for Computer Program TVSTEER, Technical Report SL-86-30. August 1986.

[WES88a] Waterways Experiment Station, Corps of Engineers, Dept. of the Army. Cone-Index-Based Estimates of Soil Strength: Theory and User's Guide for Computer Code CIBESS, Technical Report SL-88-11. March 1988.

[WES88b] Waterways Experiment Station, Corps of Engineers, Dept. of the Army. Mobility Models Utilizing Acceleration and Deceleration, Technical Report GL-88-19. September 1988.

[WES92a] Waterways Experiment Station, Corps of Engineers, Dept. of the Army. Strength Property Estimation for Dry, Cohesionless Soils Using the Military Cone Penetrometer, Technical Report GL-92-5. May 1992.

[WES92b] Waterways Experiment Station, Corps of Engineers, Dept. of the Army. NATO Reference Mobility Model Edition II, Users Guide (NRMM II), Volume I, Technical Report GL-92-?. June 1992.

[WES92c] Waterways Experiment Station, Corps of Engineers, Dept. of the Army. Simplified Army Mobility Model (SAMM). Source code (FORTRAN). Dec. 1992.

[Wong78] Wong, J.Y. Theory of Ground Vehicles. John Wiley & Sons. 1978.

[Wong89] Wong, J.Y. Terramechanics and Off-Road Vehicles. Elsevier. New York, NY. 1989.