The Newtonian Architecture for Virtual Landscapes: An Architecture, Model and Implementation

by

Martine Bruce Wedlake
BS, Portland State University, 1992
MASc, University of Victoria, 1995

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

We accept this dissertation as conforming
to the required standard

Dr. K. F. Li, Supervisor(Dept. of Electrical and Computer Engineering)

Dr. F. El–Guibaly, Supervisor(Dept. of Electrical and Computer Engineering)

Dr. V. Bhargava, Departmental Member(Dept. of Electrical and Computer Engg.)

Dr. Z. Dong, Outside Member(Dept. of Mechanical Engineering)

Dr. S. Shimojo, External Examiner(Computation Center, Osaka University)

Supervisors: Dr. K. F. Li and Dr. F. El–Guibaly

Abstract

There is much research in the literature regarding the construction of distributed virtual reality implementations. After evaluating some well–known virtual reality systems, it was determined that several problems exist that need to be solved. In particular: network efficiency, object distribution and coherency, inadequate system resource management, and overall performance.

In order to properly address these issues, a holistic design approach is taken. The entire system is examined, rather than focusing on a specific problem area (such as the human–computer interface).

The major component of this work, the Newtonian Architecture for Virtual Landscapes (NAVL), is presented to respond to the problems areas discovered. Highlights of the architecture include:

- A distributed client/server network that addressed the networking issues.
- Autonomous objects encapsulate control and object state into a single entity. Using autonomous objects avoids lengthy synchronization processes (e.g., full database locking).
- ForceLets, a novel synchronization method, minimize the network bandwidth required to keep an object synchronized at remote locations. In addition, ForceLets provide much improved synchronization of the object at the remote locations in the presence of network lag.

Implementation details of the NAVL prototype are also presented. The implementation consists of an object simulation and execution unit, rendering and collision detection unit, and network subsystem and protocols.

An evaluation of the NAVL system architecture examines the efficiency of the key architectural components:

- A bandwidth and latency analysis examines the efficiency of the distributed client/server network.
- The object distribution and coherency components are tested directly from the prototype. Profiles of actual prototype execution are used to show the efficiency gains of the ForceLet approach as compared to the commonly used stream-of-data coherency mechanism.
- The rendering and collision detection unit is tested by examining the effects on CPU utilization and frame rate with increases in the number of virtual objects.

Examiners:

Dr. K. F. Li, Supervisor(Dept. of Electrical and Computer Engineering)

Dr. F. El–Guibaly, Supervisor(Dept. of Electrical and Computer Engineering)

Dr. V. Bhargava, Departmental Member(Dept. of Electrical and Computer Engg.)

Dr. Z. Dong, Outside Member(Dept. of Mechanical Engineering)

Dr. S. Shimojo, External Examiner(Computation Center, Osaka University)

Table of Contents

List of Tables

List of Figures

List of Acronyms

API Application Programming Interface

ATM Asynchronous Transfer Mode

BSD Berkeley Software Distribution

BW Bandwidth

C/S Client/Server

CAD Computer Aided Design

CCW Counter Clock Wise

D C/S Distributed Client/Server

DEMUX De-multiplexor

DIS Distributed Interactive System

DIVE Distributed Interactive Virtual Environment

DL Display Lag

DVR Distributed Virtual Reality

EL Environment Lag

FTP File Transfer Protocol

GUI Graphical User Interface

HDLC High-level Data Link Control

HMD Head Mounted Display

IP Internet Protocol

IPv6 Internet Protocol Version 6

IRC Internet Relay Chat

ISO International Standards Organization

LAN Local Area Network

LoC Lines of Code

LoD Level of Detail

M/D/1 Memoryless, Deterministic, Single-Server Queue

MAC Medium Access Control

MUTEX Mutual Exclusion

MUX Multiplexor

NAVL Newtonian Architecture for Virtual Landscapes

NFS Network File System

NIC Network Interface Card

NPC Normalized Projection Coordinates

OPI Object Programming Interface

P-P Peer-to-Peer

PDU Protocol Data Unit

QoS Quality of Service

RSVP Reservation Protocol

SEU Simulation and Execution Unit

SMP Symmetric Multiprocessing (also Shared Memory Multiprocessing)

TCP Transport Control Protocol

UDP User Datagram Protocol

VRML Virtual Reality Modeling Language

WAN Wide Area Network

Acknowledgements

I wish to acknowledge the support and encouragement I've received by my teachers and professors thoughout my journey of learning and self–exploration.

I dedicate this work to my family and friends, who have encouraged me to pursue my ambitions.

Chapter 1

Dissertation Objectives and Organization

While computer graphics dominate our perception of distributed virtual reality(DVR), the fact is that computer graphics is only one of several research areas. Computer graphics are an essential first step to providing virtual reality(VR), but that alone will not be enough to provide effective user environments.

To improve the immersion, there may be a need to add audio or tactile feedback (e.g., force feedback) to the application. DVR systems require networking fabrics to communicate between participants.

Each aspect adds more capabilities to the VR application, but also increases overall complexity and requires more resources to be managed. Most research implementations of VR and DVR systems focus on a small subset of the whole system, preferring to off-load the responsibility of managing these extra resources to operating systems(OS) or other programming system environments.

This work takes a holistic approach — the DVR system is designed throughout the range of system activities, rather than focusing on any one part. In this way, the DVR system developer has the ability to efficiently accommodate many different resource needs within the system.

Once the resources and their needs are identified, strategies can be developed to deal with them. For DVR systems, this typically involves managing CPU scheduling, rendering speed, networking, and I/O.

This research has five overall objectives:

1. Examine current DVR implementations looking at their effectiveness as holistic solutions — where they work well, and where they do not. From this analysis, a list of specific problem areas is found. This list helps point out where the work should concentrate.

2. As part of the analysis, a benchmark (or check list) is created that can be used to evaluate specific implementations. This task differs from the task above in that this check list evaluates a particular implementation, while the previous task looked at a range of implementations. This check list specifies the most important items in a DVR system.

3. The Newtonian Architecture for Virtual Landscapes(NAVL) system architecture is developed from considering the list of problems with current implementations. This architecture is intended to address most (if not all) of the current implementational deficiencies observed with existing systems.

4. A prototype of the NAVL system architecture is developed to serve as a proof of concept and an experimental test–bed for further ideas and concepts.

5. An evaluation of the NAVL system architecture is performed using a combination of theoretical analysis, simulation, and direct evaluation of the prototype.

1.1 Dissertation Organization

This dissertation is arranged in the following manner:

Ch. 2 This chapter introduces DVR systems, provides case studies of some well–known DVR systems, and presents both the problems with current DVR systems and checklist for effective DVR systems.

Ch. 3 This chapter presents the NAVL system architecture. The architecture's major components are the network architecture, object distribution and coherency model (with autonomous objects and ForceLet coherency), and the rendering/collision detection unit.

Ch. 4 This chapter details the NAVL prototype. Major components of the prototype are the simulation and execution unit(SEU), the rendering/collision detection unit, and the network subsystem/protocol system. In addition, a route mapper is proposed that maps a path defined by reference points into a group of ForceLets. Finally, some statistics on the prototype implementation design effort is presented.

Ch. 5 This chapter evaluates the NAVL system architecture using theoretical analysis, simulation, and direct evaluation of the prototype where appropriate. The areas of evaluation include the bandwidth and latency analysis of the network architecture, evaluation of the ForceLet coherency algorithm using simulation and prototype tracing data, and an evaluation of the rendering/collision detection unit.

Ch. 6 This chapter concludes the dissertation and presents future directions for this work.

Chapter 2

Introduction

This chapter provides background information regarding DVR systems. A DVR layer representation is presented to help communicate the degree of focus for specific implementations provided as case studies.

The case studies of DVR implementations includes: VRML, DIVE, Alpha World, MR Toolkit (peers), DIS, RTIME and a few physical simulation systems. A list of DVR problems is distilled out of the process of examining the DVR implementations for their shortcomings. This list of problems gives a sense of direction for our attempts to improve DVR systems.

Some thought was also placed into describing the features that make for an effective DVR implementation. This becomes the general criteria for effective DVR systems.

2.1 Defining Virtual Reality

The literature shows many attempts to define VR. Some take a focused point of view and define VR directly. For example, Wang[49, p. 1] suggests that VR is a "highly interactive three dimensional user interface." While this definition works well for focusing attention on the specific area of interest (the user interface in this example), it fails to acknowledge the larger issues such as "Where does VR stop and Reality start?"

Burdea and Coiffet[4, pp. 2-5] examined the issue and propose that VR is the merger of three components: Immersion, Interaction and Imagination. The term imagination, here, is meant to describe the human element folded into the VR system during its development. The imagination of the scientists, engineers and artists are integrated into the final product. While immersion and interaction fit our sense of VR, the imagination component seems out of place. There are many examples of products that embody human creativity that are not VR.

Carr and England[6, pp. 1-9] take a more philosophical approach and suggest that VR is the act of synthesizing human perception. They suggest that purists at one extreme accept only technology driven definitions (e.g., computer graphics, head mounted displays). Purists at the other extreme insist that dreams and immersive entertainment (e.g., books and movies) should be included in the definition. While it is appreciated that the definition for VR has expanded beyond simple technology driven elements, it is noted that this definition is too broad to be used in a practical way.

In this work, an approach derived from Burdea and Coiffet is suggested – VR is decomposed into three elements, viewed as a pyramid (see Fig. 2.1). Unlike Burdea and Coiffet, the imagination element is replaced with computer technology. Virtual

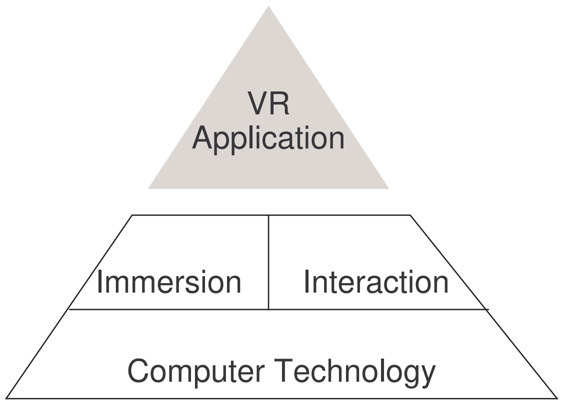


Figure 2.1: Three elements of VR

Reality is then defined as:

> *The experience the user has when he or she becomes immersed while interacting with computer technology.*

In this way, we view immersion and interaction as essential elements of VR, with computer technology as an enabler that supports VR applications. Without computer technology, we can not construct the system that provides interaction and immersion to the user.

2.2 DVR Terminology

The following terms are used throughout this dissertation:

Avatar A graphical representation of the user. It is often represented as an articulated mannequin[1]

[1] The word avatar is derived from Sanskrit to describe the coming into being of a devine entity in the shape of man or beast.

Virtual Object An object as represented in the VR world. The term is chosen to avoid conflict with object oriented programming.

Object Behavior For the NAVL system, special routines are embedded within the virtual objects. These routines are invoked upon receiving an event.

Event For NAVL, events are messages sent to a virtual object to indicate a special occurrence, such as a collision. The receipt of the event will trigger a corresponding event callback (behavior routine).

Object State For NAVL, virtual objects contain their own state information such as the object's shape, color, position and orientation.

2.3 DVR Layer Representation

Virtual Reality systems run the full spectrum; from simple single-node VR (SVR) implementations such as VRML 1.0[47], to complex multi-node DVR applications such as the Distributed Interactive Simulations(DIS) standard[14].

Borrowing from operating system design, DVR systems can be described using a layered stack as shown in Fig. 2.2. The DVR layers are: application layer, language and API layer, programming system (PS) layer, operating system (OS) layer, and the network protocol layer.

Each layer of the stack services the layer above. However, there need not be a single implementation of a service layer; in fact, we can expect to see significant competition between software developers regarding the implementation of the service layers.

Information and service dependency flow in this representation is shown by a large arrow. The arrow indicates that the application layer is serviced by the Language/API

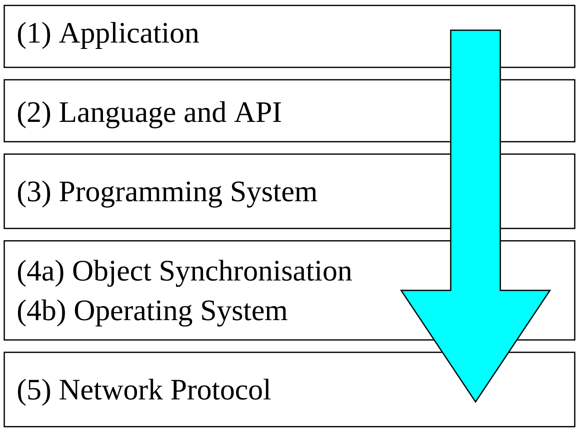


Figure 2.2: DVR layer structure

layer, and so forth. Note that when generalizing many DVR systems, some variation is to be expected. For example, it is not unexpected to find a system whereby the application has direct connections to the language/API layer and to the OS layer. It is unusual, however, to see the implicit ordering violated (e.g., a DVR PS layer that services requests for the OS layer).

Layer 1: Application

The application layer represents the set of DVR applications. There are many variations of applications possible, the following is a short list of some application areas being implemented for VR and DVR systems (see[4, Ch. 8] for a comprehensive survey, and more recently[18]).

Entertainment is likely the largest growing market segment of VR applications. Given our earlier definition of VR, many home computer games are small scale VR systems, such as car-racing or airplane simulators.

Physical Simulators are much more complex versions of the entertainment class simulators. In addition to real-time computer graphics, physical simulators provide feedback to the user in terms of force (e.g., through mechanical actuators), typically in response to a dynamics model. Futhermore, tighter tolerances to the real world are used for the dynamics models. For example, airline physical simuators are sophisticated enough that pilots can upgrade their license via simulator training alone[26].

Scientific Simulation and Visualization use immersive display techniques to allow the scientist a better view of the phenomena measured. For example, VR techniques are used in the visualization of electro-magnetic fields for micro devices[29].

Productivity Enhancement Tools allow people to work together better. Video conferencing, for example, allows meetings to be held without flying people in from around the country. People can participate without leaving their office. DVR allows the participants to be immersed in the same cyber environment while performing complex tasks[20].

Layer 2: Language and API

Applications employ DVR system services with either a set of languages (possibly extensions of existing languages) or application programming interface (API) calls. The advantage of a language/API layer is the abstraction and encapsulation of services.

Example of languages include the Java-3D[31], and Inven/TCL[16] language extensions, and the purpose-built VRML language[47]. For DVR systems, common APIs include OpenGL for graphics[37] and WinSock[44] for networking.

Layer 3: Programming System

The DVR programming system layer handles requests made through the language extensions and API layer, providing infrastructure for managing distributed VR environments. An example is the Java Virtual Machine[28].

Layer 4: Object Synchronisation and Operating System

The object syncrhonisation and operating system services layer maintains low-level machine services such as synchronising virtual objects, CPU scheduling, memory allocation, local I/O, and networking. Most DVR implementations are built on top of an existing operating system like UNIX, Linux, or Windows-NT.

Layer 5: Network Protocol

The protocol layer describes the composition of the networking connections; the bits on the wire. For example: TCP/IP[45], Network File System[11], Network Information Service[11], Domain Name Service[11] and Simple Network Management Protocol[45].

2.4 Case Studies of DVR Implementations

The following is a summary of some of the more prominent DVR systems found in the literature and how they fit into the DVR layer structure; the DVR layer numbers involved are listed parenthetically. It appears that many DVR systems are focused on a small subset of the DVR layers.

In particular, it is evident that existing implementations leverage the OS services from their native OS (e.g., Windows-NT or UNIX). From this list of case studies, the reader will notice that none of them implement services in the OS layer (layer 4).

2.4.1 VRML(2, 5)

The VRML specification was created to provide a standard framework for the development of VR languages over the Internet (Version 2.0 – ISO/IEC 14772)[47]. The initial version, 1.0, specifies a textual language for describing static virtual environments. With VRML 1.0, authors can describe reasonably complex scenes using various shape primitives (e.g., cone, box, sphere, plane) and lighting primitives (e.g., directional, ambient, point light).

The second version of VRML, 2.0, was adopted from the SGI Moving Worlds[42] proposal. Version 2.0 specifies mechanisms to add behavior routines to the graphic objects. Graphic objects in VRML 2.0 can issue input and output events that may be used by other objects (or scripts) in the scene. The scripting facility allows for a programmed behavior to generate and receive events. For example, a user activating a touch sensor generates an event that starts up a script. The script may, in turn, generate other events as a result of its execution.

There are significant shortcomings of the standard; for example, the VRML specification does not allow for multiple users in the same environment. In addition, there needs to be a tighter reign on the specifications to support multiple users appropriately. In particular, the VRML 2.0 specification explicitly mentions that the passage of time is determined by the browser, which may decide to speed up or slow down the virtual world at its discretion. To support multiple users in the same virtual environment, the passage of time must be closely synchronized between distant hosts. In response to this problem, the Living Worlds[48] group has proposed a collection of changes to the VRML specification to allow multiple users via explicit points of synchronization.

In addition, the networking architecture is client/server based. The central VRML server must be large enough to handle the expected load, which can be considerable. For the moment, the load activity is limited to downloading VRML documents, but the Living Worlds extensions would require that the VRML server also synchronizes virtual objects, thereby greatly increasing network traffic and CPU loads in the server.

2.4.2 DIVE(2, 3)

The DIVE DVR system, developed to experiment with distributed virtual worlds, is fully distributed (i.e., every node connects to every other node in the network) using reliable IP multicast to transmit messages to all nodes[5]. Virtual objects are replicated on each node such that every node in the system has equal access to every virtual object. Because there is no centralized server node, a change made to any replica of a virtual object will affect all other replicas. This is accomplished with the emulated shared memory. The local node updates state information for a virtual object, and the DIVE networking subsystem replicates the change using the multicast IP. Mutual exclusion (MUTEX) operators serialize simultaneous updates to the shared virtual objects.

Controlling the state of the virtual objects is a collection of event driven behavior routines written in a hybrid DIVE/Tcl language. These routines are started when an event is received such as a user interaction signal, timer or collision.

The user's representation in the virtual world, called an Avatar, is integrated into DIVE at a fundamental level. Many VR systems implement avatars as a regular virtual object that is tied to the user's position; when the user changes position the avatar virtual object responds in kind. In DIVE, however, the avatar object is a fundamental part of the user's state; for example, the rendering process shows the view from the avatar's left eye; or both eyes if using a head mounted display(HMD).

The position of the avatar dictates what the user sees.

Since DIVE's development in the early '90s, it has been shown that more sophistication is required to deal with interactive distributed virtual environments. Particular criticism is placed on the decision to use an emulated shared memory communication model with MUTEX protection. This model greatly increases the latency to update virtual objects. The choice of peer-to-peer networking with multicast IP causes concern for those who do not have system administrator privileges to enable the multicast IP mechanism. DIVE can be set up to emulate multicast IP via a series of UDP transmissions, however the networking load grows to order n^2 packets per time cycle. It quickly becomes apparent that an efficient network architecture is needed.

2.4.3 Alpha World(1)

Alpha World, a beta release of Worlds Chat and now the property of Circle of Fire Studios, Inc., is a multiple-user VR system for socializing - very much like a VR form of IRC (Internet Relay Chat)[9]. A special browser, used to enter the Alpha World system, supports the rendering of the three dimensional graphics and the communication protocols to interface with the Alpha World server.

Users wander around the Alpha World city looking for interesting places to meet and talk. The beta version limited talking to keyboard entry, however the production version allows for voice transmissions. An interesting feature is the ability of users to build virtual structures in the Alpha World city. The city is always changing and evolving, providing an interesting backdrop for conversations.

The networking architecture is based on the client/server paradigm where the communication protocols follow two independent networking mechanisms:

1. Static world descriptions are downloaded using FTP. These descriptions are

similar to VRML in that they are ASCII based.

2. Interaction between users and the virtual environment is handled with a proprietary communication protocol.

These mechanisms naturally segment the communication flow into a low priority bulk download packets, and high priority synchronization packets. As the user moves through the virtual environment, virtual objects are continuously coming into view. The delay while the virtual object description file is being downloaded is quite apparent to the user. However, when the user interacts with the avatar of another user, the interaction response is very quick. This division of networking bandwidths is very well suited to virtual society applications, where the focus is typically on the chat aspects rather than the virtual environment.

The Alpha World system works very well for the virtual society application. The emphasis on interaction with people rather than virtual environment keeps people involved with their conversations at the expense of environment lag.

As a general purpose DVR system, however, the Alpha World system has some significant shortcomings. In particular, Alpha World uses a client/server networking architecture. As will be seen in Ch. 5, the client/server networking architecture does not distribute the network load; all network traffic funnels through the server machine. This implies that the Alpha World server must have a very wide Internet bandwidth connection, and it must have enough CPU power to support all the clients. As the Alpha World city grows, the available bandwidth and CPU will not be enough to compensate.

2.4.4 MR Toolkit with Peer Package(2, 3)

The MR Toolkit is a collection of libraries to build simple VR applications, initially developed to experiment with HMDs and VR input devices[49, 38, 39, 40]. The system is modular, so additional subsystems may be deployed at will. The base MR Toolkit is a collection of libraries and device drivers to support research into real-time interaction for a single user virtual environments. The Peer Package[38] provides support systems for multiple MR Toolkit applications to communicate using UDPs. The communication subsystem allows MR Toolkit processes to update their peers with current local device data – essentially checkpointing their state onto the set of peer MR Toolkits. In addition, the Peer Package allows for application specific communication between peers using a simulated shared memory communication model.

The connection topology is peer-to-peer; limiting the scope of the peer package to approximately five distributed users[38]. To compensate for the lack of available bandwidth, the Peer Package uses a system whereby virtual objects are owned by MR Toolkit processes. When a process owns the virtual object, the simulation is performed locally and the effects are transmitted to all other participants, but at a reduced update rate (e.g., every fifth simulation result for the hand-ball example in [38]). The overall effect is that the owner of the virtual object sees it move with much better fidelity than the remote users.

2.4.5 DIS(5)

The DIS standard[14] describes a protocol for managing warfare simulators over a wide range of communication media. The simulation exercise is separated into a collection of simulation applications; each application is responsible for representing the virtual manifestation of one or more simulation entities (e.g., manned vehicle simulator).

These simulation applications are distributed throughout the network with a standardized protocol for communicating ground truth data. Changes in state are determined by the controlling simulation application, and the perception of events is determined by the receiving application. For example, it is possible for targets to be obscured by smoke or terrain. Since the ground truth is transmitted throughout the network, the receiving application is responsible for not showing the obscured targets to the user.

To reduce network load, DIS uses a dead reckoning protocol. Each simulator maintains two states: the internal (i.e., real) state, and the approximate (i.e., dead reckoning) state. Only the dead reckoning state is transmitted to receiving applications. The dead reckoning state is evaluated according the dead reckoning model chosen, and when the dead reckoning state diverges from the internal state the dead reckoning state is updated to match the internal state. Messages are only transmitted to the receiving applications when the dead reckoning state changes.

There are nine dead reckoning models defined in the standard (and an additional two models are suggested), with the following model being the most general (other models are lower-order simplifications):

$$P = P_0 + V_0 \Delta t + \frac{1}{2} A_0 \Delta t^2 \tag{2.1}$$

$$[R]_{w \to b} = [DR] \, [R_0]_{w \to b} \tag{2.2}$$

where P, V, and A represent position velocity and acceleration; $[R]$ and $[DR]$ are the orientation and dead reckoning matrix.

Dead reckoning provides a very nice beginning step, however it is too limited for general purpose use. The quadratic form shown above is excellent for munitions trajectories; but it can not be used as a basis function for generating complex paths through space-time. Whenever a virtual object must follow such a path, the internal state and dead reckoning state will constantly be out of sync; thus requiring a refresh. Also, there is a computational drain associated with having to maintain two states (and converting from the internal to the dead reckoning state).

2.4.6 RTIME Interactive Networking Engine(2, 3)

The RTIME Interactive Networking Engine[36] is an elementary object based programming system, developed to support multi-user games over the Internet. The engine is client/server based with capabilities for decoupling the environment updates from the graphic refresh rate, global time synchronization, virtual object filtering (based on object type and/or distance). The overall purpose is to provide fair access to the server for all clients (e.g., players) in a multi-user game.

The engine API provides calls for: managing virtual objects (e.g., creation/destruction), updating virtual objects (e.g., assigning position, velocity, and acceleration), finding objects (e.g., finding objects by name or association).

As with Alpha World, the client/server connectivity paradigm limits the scalability of this implementation. To accommodate larger virtual worlds, significant upgrades of both the computational and communication resources are required.

2.4.7 Physical Simulation Systems(1)

Robotics and automation experts have been using dynamics simulators for many years, generally to model physical equipment. The Newton general-purpose dynamics simulator[12] was developed to investigate many-degrees-of-freedom articulated objects (i.e., robotic limbs). A special computer language was developed to allow the user to describe physical and geometric attributes of the objects to be simulated. A binding element called a hinge, defines how objects are attached (e.g., spherical joint or pin joints). The generated system of Newton-Euler equations is then evaluated for each time step to perform the simulation. Collisions are handled using a relatively simple impact resolution scheme: at the point of impact, the geometry of the objects is consulted to determine the resulting changes in velocity and angular momentum.

In addition to modeling real-world devices, the dynamics simulators approach has been increasingly popular among the animation community. The kinematic clones system[58] models articulated figures (e.g., cartoon characters) using a kinematics model. The essential idea is to model an animated figure as a mechanical puppet with springs and dampers to provide a natural flow to the animation. This requires fewer "key frames" that the animator must be involved with. The kinematic clone is simulated using a dynamics simulator similar to the Newton system described above.

We may also gain inspiration from systems involving remote space-born robotics. Hirzinger[23] suggests a state-space model for a telerobotic system. The state-space model incorporates the transmission delays to and from the satellite, thereby allowing for the computation of the predictive state of the robotic arm (see Fig. 2.3). The operator, on the ground, is shown a computer graphic representation of the predicted position of the robotic arm that is manipulated in the operator's real-time; the controls are transmitted to the remote robot and the results (e.g., position) are transmitted back to Earth to update the state-space model.

The CPU requirements for a dynamics simulator, kinematic clones, or state-space approach are too high for a real-time graphics display, even with today's high performance CPUs. For use in interactive applications like VR, we need to strike a balance between the physics based modeling and what can really be achieved on contemporary computers.

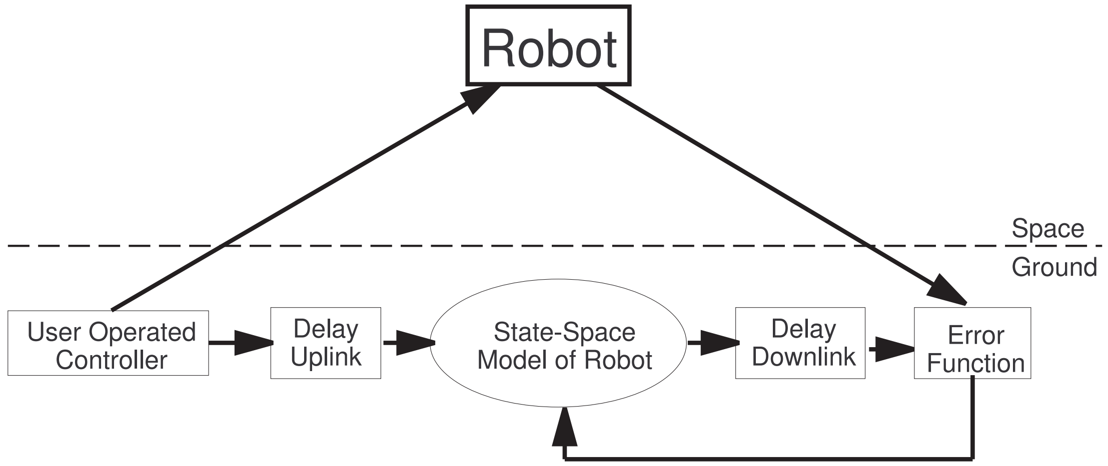


Figure 2.3: Simplified Hirzinger model for telerobot. The State-Space model of the robot is used to drive a 3-D graphics display

2.5 Problems with Existing Implementations

The preceding case studies show that many of the current DVR systems have inherent difficulties. The following issues have been identified: (1) network efficiency, (2) coherency models, (3) virtual object distribution strategies, (4) inadequate system resource balancing, and (5) overall performance. It is argued that many of these problem areas are ultimately caused by inadequate system architectures, and lack of holistic design.

2.5.1 Network Efficiency

Nearly all DVR systems use either multicast Peer-to-Peer (PP_{mc}), unicast Peer-to-peer (PP_{uc}) or client/server (C/S) network architectures. The abbreviation, P-P, will be used to indicate a peer-to-peer network architecture with an unspecified transmission mode.

$\mathrm{PP}_{\mathrm{mc}}$ is a one-to-many transmission mode in which a single message is sent to multiple network destinations[15, p. 4]. All nodes may send messages over the multicast channel to communicate with all other nodes with a single message.

$\mathrm{PP}_{\mathrm{uc}}$ is a one-to-one transmission mode in which a single message is sent to a single network destination[15, p. 6]. Multiple messages are sent in order to cover multiple destinations.

C/S is when a special node called the *server* is responsible for handling requests made by the remaining nodes (called *clients*). The clients are only permitted to communicate to the server, while the server communicates to all the clients using either a unicast or multicast protocol.

Several problems exist with P-P and C/S architectures:

1. The bandwidth required by $\mathrm{PP}_{\mathrm{uc}}$ grows by the square of the participants.
2. Multicast suffers from the problem where clients receive messages that they are not interested in, thus leading to wasted bandwidth.
3. Multicast networks require special setup in order to be deployed. Many businesses and universities do not allow multicast transmissions due to the extra bandwidth required.
4. The C/S model exhibits a severe bottleneck at the server node. Both communication resources and computation resources must be scaled according to the number of participants.
5. The C/S model also suffers from low availability due to being a single point of failure. The entire system comes down in the event the server crashes.

An interesting example of a DVR system that does not use either P-P or C/S is the WAVES architecture[25]. This architecture uses special nodes called *message managers* to partition the network and coordinate between remote regions of the network. Neither a P-P or C/S paradigm is suggested, instead a "tree-like" structure is built up with message managers communicating between "sub-trees".

2.5.2 Object Distribution and Coherency Models

Proper management of virtual objects in a DVR system is critical when the numbers of virtual objects and hosts in the DVR system grows. The literature has shown many approaches to managing virtual objects on multiple hosts; for example:

Laissez-faire or optimistic concurrency control[11] assumes that no interaction is required between virtual objects and corrects the assumption when proven wrong (such as a collision event). Often, a special "synchronize now" call is made available to the application programmer to be used at his discretion.

The laissez-faire approach can only work when there is little need for synchronizing virtual objects. Anytime virtual objects are used by multiple parties, the system breaks down.

A **Centralized Database** maintains the object's state information in a single server, thereby allowing the server to easily synchronize the state. This is used by the client/server systems such Alpha World[9]. The centralized database approach requires all requests to be funneled into a single resource (e.g., network router, or server machine). The lack of distribution causes bottlenecks within the network fabric which are only addressed by deploying larger and faster systems (expensive). In addition, the centralized database represents a single point of failure. If the server machine goes down, then the entire database is unavailable until restored.

Object Ownership Transferal systems replicate the database on all hosts, but only one host has ownership over an object at any point in time. The MR Toolkit describes object ownership transferal[38]. It is difficult to see where generalized use for ownership transferal mechanisms can come into play. This scheme is only appropriate for applications where a natural ownership is exhibited, such as a game of squash (specified in [38]).

Locking the virtual world database enforces consistency by restricting access to the database itself. The DIVE DVR[5] system locks the database with mutual exclusions. Locking provides a good mechanism for ensuring that transactions are carried out sequentially, however using a locking mechanism for distributed applications involves a lot of network traffic and involves long time lags. Typical distributed locking schemes require messages to be sent repeatedly to all hosts[11]; making it nearly impossible at current network latencies to have real-time interaction.

Dead Reckoning is used by the DIS standard[14] to reduce network load. Dead reckoning provides a mechanism to describe behavior of a virtual object over time without the need for a steady stream of update messages from the simulator process. Unfortunately, the behavior is limited to simple, static mechanisms (e.g., munitions trajectories and steady state velocities). The choice to have two simulation models (the real application's simulation model plus the dead reckoning model) requires extra work on the part of the application to translate from the application's model to the dead reckoning model. Unfortunately, the model used by the dead reckoning protocol is not sophisticated enough to host generic application objects.

2.5.3 Inadequate System Resource Management

Many DVR developers leverage operating system (OS) services to manage system resources. While many of the attributes of DVR systems parallel those of distributed operating systems (e.g., Amoeba[8, 11]), there are two main differences between them:

First, distributed OS do not require high levels of immersion or interaction. For example, the Network File System (NFS) is a common distributed OS component. However, the NFS is not suitable for supporting hundreds of hosts manipulating the same files; both in terms of file integrity and performance.

Second, many distributed systems emphasize accuracy and data integrity; often going to long lengths to ensure transactions are atomic[8, 11]. Distributed virtual reality systems, however, are typically concerned, primarily, with providing immersion and interaction to the participant; data integrity comes second.

These differences are often a result of the manner for which the system manages the resources (e.g., early vs. late binding of resources).

2.5.4 Overall Performance

It is inherently difficult to assess the relative performance merits of one DVR implementation over another. One reason is because of the "apples and oranges" comparisons that inevitably arise. For example, we would find it very difficult to compare the MR Toolkit to the DIS protocol - they are just too different. Another problem is the difficulty in defining performance in the first place.

Traditionally, applications are gauged by their elapsed time. For example, a compiler that finishes in 23 minutes is faster than one that compiles in 38 minutes. Elapsed time measurements work well for sequential programs.

DVR applications, however, are not sequential applications. Instead, they are driven by the user clicking the mouse or gesturing with the DataGlove. Maintaining a high degree of interaction is often the most important task for DVR system. The degree of interaction can be improved by increasing the frame rate, reducing time lags when operating virtual controls, and using more sophisticated rendering models (e.g., texture maps and shading models).

Furthermore, managing the fidelity of the environment as a whole takes on a new importance. For example, a fast frame rate is considered important for users to feel connected to the virtual world; however, a fast frame rate with wire-frame graphics might not be as appropriate as a moderate frame rate with fully rendered graphics.

Even though it is difficult to measure the performance of DVR systems, it is still important to provide a suitable platform for executing DVR applications. Taking these issues into account, the following criteria are suggested for measuring the performance of a DVR system:

Synchronization Fidelity is the measure of the disparity between hosts' views of the same virtual world. To support interaction between users, their views of the virtual world must be as close as possible to each other.

Environment Lag is the amount of time it takes to notify a remote host of a change made to a virtual object. A long environment lag promotes confusion within the virtual environment as objects become out of sync with each other.

Display Lag is the amount of time it takes to draw the scene for the user. Long display lags promote simulator sickness, and general confusion for the user.

DVR Capacity is a measure of the maximum number of hosts the DVR system can reasonably support. Ideally, the DVR system will scale by the number of hosts.

Graphic Complexity indirectly measures the quality of the graphic model. Higher graphic complexity generally reflects a higher sense of realism. For example, more polygons or sophisticated rendering algorithms create more realistic scenes.

2.6 General Criteria for Effective DVR Systems

A large contribution to the success of a DVR implementation is the effect on the display and environment lags. Display lag, the delay between refreshes, affects the user's comfort during the virtual experience because a long display lag can lead to the onset of simulator sickness[50, 6]. Environment lag, the delay between object updates within the VR world, affects the user's ability to interact in the virtual world. A long environment lag prevents the user from being able to anticipate an object's trajectory thereby promoting frustration. Studies indicate that the display lag should be less than 80 ms to maintain hand-eye coordination[6]. Unfortunately, similar studies for environment lag have not been located.

A list of requirements for DVR is provided below. Requirements affecting display lag are tagged with **DL**, and those that affect environment lag are tagged **EL**.

2.6.1 Better Graphics (DL)

Fast rendering speed is essential for an acceptable refresh rate (e.g., 10-40 Hz). Depending on the scene complexity this may be easily obtained with the main CPU, or external graphics accelerator cards may be required. This requirement also affects the visual realism. High quality computer graphics (e.g., ray tracing) are also CPU intensive[19], thus a balance must be made between the need for good quality graphics and the available CPU budget.

Scene rendering rates are greatly affected by architectural issues such as the choice of CPU and graphics library. The DVR designer has some control over the refresh rates by selecting different rendering algorithms (e.g., using texture maps and complex lighting models require more CPU power) and limiting scene complexity (e.g., fewer polygons).

2.6.2 Fast Networking(EL)

Fast networking (low latency and high bandwidth) keeps the distributed nodes synchronized. Low latency is important for quick transmission of small requests and responses, while a high bandwidth is necessary for transmitting large chunks of data, such as when loading the virtual world.

2.6.3 Synchronized Virtual Events(EL)

This ensures that when two people, in the real world, manipulate an object simultaneously (or nearly) the object will react to both interactions. For the virtual environment to model this behavior, architectural support must be built into the DVR system.

2.6.4 Ease of Use

It is important for people to have easy to use interfaces for developing virtual objects in the DVR system. Some systems address this by providing means to translate files developed using professional CAD programs, other systems provide a CAD application written for the DVR system. An easy to use DVR system enables faster deployment and quicker adaptation, which is essential for general acceptance.

In addition, the system must also be easy to design applications on it. Easy to use programming environments encourage the rapid deployment and quick adaptation of many applications.

2.6.5 Autonomous Virtual Objects

Autonomous virtual objects enable the objects to react to the virtual environment without resorting to monolithic solutions embedded within the system itself. This increases flexibility of the virtual world because the behavioral component can be extended to provide application programming *within* the virtual world.

2.6.6 Heterogeneous Computing Platforms

People with a diverse selection of computing hardware should be permitted to interact together. For example, some users in a DVR may have modem connections (i.e., low speed), while others may have low resolution monitors. This requirement, however, introduces many difficulties that must be overcome. For example, a user with a slow connection speed but fast CPU will need to emphasize the local computation ability to overcome the slow connection speed. The opposite is true of fast connection but slow CPU bound machines. This balancing of CPU to networking resources introduces a new set of constraints that are critical to be solved in order for participants in the virtual world to get the most out of their hardware.

The team that created the Cyclades compute network argues that heterogeneity is an advantage[2, p. 11]:

It allows one to get the best from each system, since it removes the constraint of having one single type of system for all applications. Machines can be dedicated to the work they are best adapted to (e.g., business oriented, scientific, data base).

2.7 Chapter Summary

In this chapter we began with a discussion to define VR. After examining some of the current definitions, we came up with three essential parts of VR: immersion, interaction, and computer technology.

A series of case studies was then presented to give the reader a feel for the state of the current art of VR design. In particular we examined the: VRML, DIVE, Alpha World, MR Toolkit, DIS, RTIME, and Physical Simulation Systems.

The case studies demonstrated several problems that exist with contemporary DVR systems. These problems were identified as: network efficiency, object distribution and coherency, inadequate system resource management, and overall performance.

Finally, we itemized the list of items that are required for an effective DVR system – the criteria or benchmark: better graphics, fast networking, synchronized virtual events, ease of use, autonomous virtual objects, and heterogenous computing platforms.

Chapter 3

NAVL System Architecture

The NAVL system architecture attempts to address (from the functional point of view) the problems of DVR systems discussed in Ch. 2.

The major components of the NAVL system architecture are:

1. A network architecture that reduces bandwidth and latency.
2. An object distribution and coherency model that provides better user-to-user synchronization.
3. A combined rendering and collision detection system that takes advantage of the shared elements of their respective components.

3.1 Objectives of System Architecture

The most important objective of the system architecture is to address the problems found in Ch. 2. An *engineering approach*[1] to these problems is used to develop robust

[1] An engineering approach is different than a scientific approach in that it is concerned with efficiently applying technologies to construct products, whereas scientific approaches are often used

solutions. Usually, this involves the integration of common off-the-shelf components. However, on occasion, this also involves the creation of new technologies as will be seen in the coming sections

In addition, as part of the engineering approach we desire to use holistic design principles when building our DVR system. As was seen with the case studies, many contemporary DVR systems are focused on a narrow range of services. An effective DVR system of practical value needs to integrate a wide variety of services. This "wide angle view" requires a holistic approach.

The final objective is to find solutions that scale well. Some argue that detailed software engineering is not required with current software products because hardware capabilities are increasing so quickly. In opposition to this view, it is felt that tremendous benefits can be realized if the software is well engineered. In particular, a well engineered distributed system will scale better and require less hardware (i.e., low cost) to function at a comparable level of performance. The extra care placed in the design phase is very small compared to the cost of redesigning a non-scalable system.

NAVL is best able to support applications with little synchronisation requirements (e.g., no atomic transactions). Applications such as tele-surgery would not be suitable to NAVL because the NAVL system has no mechanism in place to "roll back" from an operation. Due to the uncertainties of network latencies and the synchronisation mechanism used by NAVL, it is possible for remote hosts to be temporarily out-of-synch, which might cause an operator to make an error in his or her actions.

---

to discover or prove theories.

3.2 The NAVL Architecture at a Glance

The NAVL system architecture has three primary components: (1) the network architecture, (2) the object distribution and coherency architecture, and (3) the rendering and collision detection architecture. The NAVL architecture exhibits the following capabilities:

Autonomous Virtual Objects allow an encapsulation of the virtual object that may be moved as needed to balance the CPU and networking loads.

Distributed Client/Server Network Architecture reduces bottlenecks within the network by spreading the bandwidth throughout the network, while simultaneously reducing the load to the network between peers as compared to other network architectures (discussed in Ch. 5).

Master/Slave Virtual Object Distribution and Coherency Model splits the virtual object into a single master object and several slave objects. The master object is responsible for the state of the object while the slave objects represent the local manifestation of the object. The slave objects execute high-order commands called ForceLets that describe a path through space for the object to follow.

Local Simulation of Virtual Objects by Slave Objects reduces network bandwidth requirements by executing much of the virtual object's actions local to the user rather than at a remote location (e.g., server).

3.3 Network Architecture

The purpose of the network architecture is to enable multiple participants and allow the sharing of computer resources (e.g., CPU and storage). As seen in Ch. 2, problems with existing network architectures are often related to the network's topology. Client/Server systems do not scale well to increasing numbers of clients due to a severe bottleneck at the server node. Peer-to-peer systems also do not scale well to increasing participants because the number of messages sent through the network infrastructure grows quadratically.

The target network environment for NAVL is a large-scale network with many local area networks (LAN) connected by a wide area network (WAN). Routers connect the LANs to the WAN interconnect.

In many network deployment, such as an office environment, people working on the same areas of interest are often placed into their own LANs. Generally, this is a side-effect of the natural evolution of the network within the organization. For example, the engineering department is often located in a different floor (or even building) from the sales and marketing departments, thus creating a natural boundary for LANs within the organization.

The NAVL DVR network architecture leverages this natural partitioning of network resources, employing a dual-region network architecture. The LAN in NAVL uses a broadcast transport mode such as found with Ethernet, while the WAN is a collection of peer-to-peer channels that link up the LANs.

Using a broadcast LAN allows participants to snoop on messages. Many messages sent in the DVR system affect virtual objects distributed between many participants; in the situation where the majority of the affected participants are contained within the LAN, a lot of bandwidth can be saved by leveraging the broadcast medium.

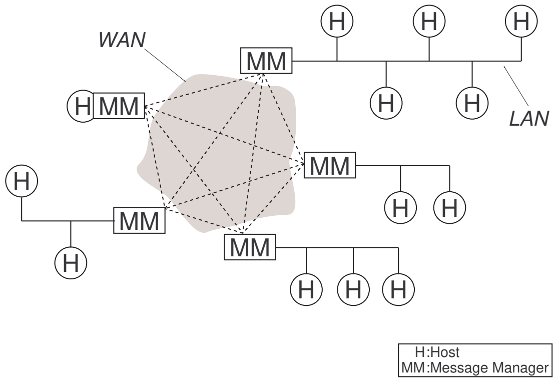


Figure 3.1: Distributed client/server network architecture

The peer-to-peer connections use quality of service (QoS) technologies whenever possible, such as available with ATM, IPv6, and RSVP (see [3, 43, 45]). The QoS capabilities allow systems to specify the maximum acceptable latency and jitter within the packet switching networks.

Quality of service is helpful for the DVR system to ensure a consistent view of the shared virtual environment to remote users. However, it is not always available. For this reason, the NAVL system architecture employs mechanisms to reduce the effect on the user from latency and jitter (i.e., ForceLets).

Figure 3.1 shows an example deployment of the NAVL network architecture called Distributed Client/Server. The WAN is shown as a collection of peer-to-peer channels between the LANs, where the LANs are shown as a collection of NAVL hosts connected to a common broadcast medium.

Between the two regions are special server hosts called message managers (borrowed from the WAVES architecture[25]). The message manager has four tasks:

1. Router between the LAN and WAN. This is similar to the tasks performed by

existing IP routers, albeit at a different network layer[2].

2. Filter messages going to and from the WAN. By filtering messages going to and from the WAN, the message manager has control over the loading of the rest of the network. For example, the engineering department may not wish to export their virtual objects outside their departmental LAN.

3. Directory Service allows a quick lookup for virtual objects or participants in the network.

4. Quality of Service capabilities within the message manager enable it to select different QoS service levels. This may be useful if the application needs to briefly increase its QoS resources for some purpose, or to drop its QoS resources.

The distributed LAN architecture described in this section can be thought of as a combined peer-to-peer and client/server model.

3.4 Object Distribution and Coherency

The term "distribution" used here means the overall mechanisms for arranging virtual objects in the network such that they can be accessed by multiple participants. These mechanisms often include networking architectures (e.g., distributed client/server as seen above) and access algorithms (e.g., locking and two-phase commit).

Coherency, however, refers to the mode for which the virtual objects are updated and kept synchronized. For example, the DIS standard uses dead reckoning to move objects through space.

[2]NAVL routing is at OSI layer 7 while existing routers are at OSI layer 3.

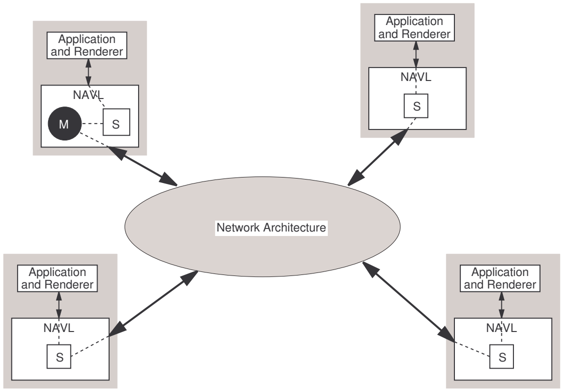


Figure 3.2: NAVL architecture elements

An example of object distribution can be seen in Fig. 3.2. This diagram shows four computers connected to a network architecture.

Each computer has a NAVL subsystem (called the NAVL host) with an application connected to it. Within each NAVL host there can be virtual objects (the disks and boxes). The dark disk labeled 'M' is the master object, and the boxes labeled 'S' are the slaves. Within the NAVL host, dashed lines indicate information flow between the network, the objects, and the network architecture.

Users interact with applications, which are shown within the NAVL host. The user interacts with the application using traditional I/O devices (e.g., mouse, keyboard and 3D display). This work is primarily interested in the services provided up to the application layer, so often we will consider design decisions' effects on applications rather than users–although we will consider the effects on the user when warranted.

Applications interface with slave objects located on the same computer. Slave objects, in turn, communicate to the master object through the networking architecture (or directly in the case where the slave is on the same computer as the master). The master object is responsible for performing operations requested by the application.

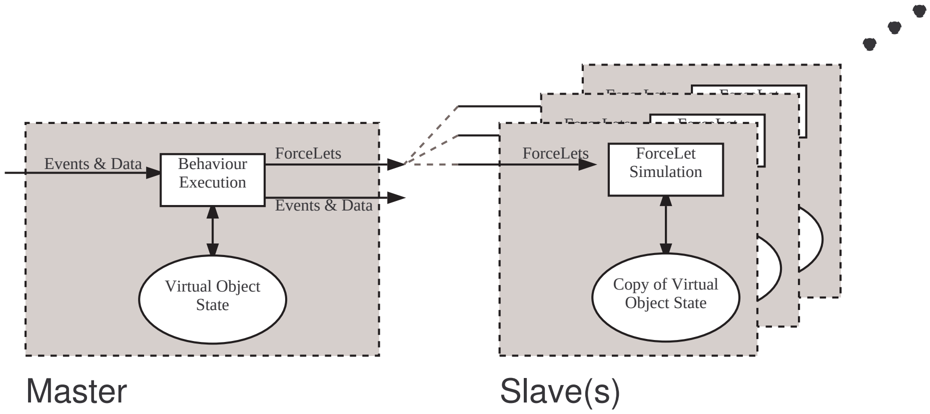


Figure 3.3: NAVL implementation of virtual object

The results of these actions are sent back from the master object to all the slave objects (including the slave object the user is interacting with).

In Fig. 3.2, the master object issues special messages called ForceLets to update the state of the slave objects. ForceLets specify how a virtual object should move through space. They are tied to a synchronized global clock in each NAVL host to ensure that the effects of simulating the ForceLet produces the same results to all participants.

To illustrate the master and slave objects, Fig. 3.3 shows a diagram of the virtual object with a single master and one or more slaves, separated by a node boundary (e.g., network interconnect). The master object is responsible for executing event driven behavior routines, while the slaves updates their copy of the object state by simulating ForceLets.

3.4.1 Autonomous Master/Slave Replication

As seen in Sec. 2.5.2, many commonly used distribution mechanisms have either a too conservative access mode (e.g., locking) or are too lax (e.g., laissez-faire) resulting in poor virtual object consistency.

In NAVL, applications create and free virtual objects. Currently, there are no provisions for ownership, thus any application can free any virtual object (even those created by another application). To provide autonomy and efficient object distribution, the virtual object is separated into a single *master* object and a group of one or more *slave* objects.

The master object holds the one true copy of the object's state. With client/server distribution, the master objects are located on the single server machine and accessible to client hosts and applications through special interfaces. In NAVL, however, the master objects are not required to reside on a single host; applications can create master objects in the same host it resides.

Copies of the master object, called slaves, are replicated on demand by application programs. These slaves live on the local machine, close to the application that requested them. The slave objects are continuously updated with new information by the master object as the virtual state changes. In this way, the application program does not need to worry about the master object at all; instead, it interfaces directly with the slave object. Outside the view of the application, the slave object is continuously sending messages and being updated by the master object.

The master objects are responsible for executing behavior routines embedded within the master object itself. These behavior routines are how the master object responds to requests and events within the virtual environment. For example, a master object's behavior routine can be programmed to respond to a user event such as a mouse click.

Events can originate anywhere within the system; for example, an application can send messages to virtual objects to issue its requests, the NAVL host can send messages when collisions are detected between virtual objects, and virtual objects can send messages to other virtual objects.

Behavior routines operate in a time-slice manner. There are no restrictions to duration or time of execution; they run whenever and for however long as required. However, they can be preempted when necessary in order to allow high priority tasks to run. This allows for complex requests to be handled within the behavior execution environment.

This level of freedom does require more rigor on the part of the developer however. For example, in order to support a generalized execution environment we need to ensure that the behavior routines are thread-safe and re-entrant. This allows the same behavior routine to be executed.

The intention is to provide a generalized distributed computation environment that supports a multi-threaded and symetric multiprocessing(SMP)[60] capable execution environment.

Symmetric multiprocessing systems connect two or more CPUs to a common (shared) memory system, often using a bus topology. The memory bandwidth is designed high enough to support multiple memory accesses with minimal negative effects. The chief advantage is that multiple tasks can be executed simultaneously while having a very fast access to the same data structures.

3.4.2 ForceLet Simulation

The stream-of-data and dead reckoning approaches are two very common approaches for updating virtual objects in DVR systems. The stream-of-data approach refers to the technique of sending a constant stream of position data through the network to the remote hosts.

Dead reckoning uses two object environments: the real-world environment that holds the correct positions for all virtual objects and a lower-order approximation environment. A single simulator application maintains both environments.

When the two environments diverge by a specified threshold, the approximation environment is reset to the current real-world environment. The reset operation requires an update of the virtual objects used by the remote hosts.

NAVL improves upon the dead reckoning approach by merging the real-world and approximation environments. The update mechanism is made powerful enough to support a large number of real applications.

The underlying update mechanism is based on ForceLet evaluation. A ForceLet is a network message that describes the application of a force to the virtual object over a specified time interval. Some useful (but not necessary) properties of ForceLets are:

Combinable: ForceLets should be made such that they can be combined together to extend their capabilities.

Localized ForceLets should have finite life spans. This allows remote systems to remove them when they are no longer active, rather than waiting for a signal to remove the ForceLet.

Fast Computation: The ForceLet should impose minimal overhead on the NAVL host to compute.

Meaningful: The ForceLet should map to a natural phenomenon, such as moving an object from place to place. This improves the developer's understanding about what is occurring and promotes an easy to use system.

The notation for a ForceLet is $F_D(A, \tau, T; t)$ where D is the domain (f for force, v for velocity or d for displacement) and the parameters: A for amplitude, τ for start time, and T for duration. The point in time is specified by t and is synchronized to all hosts using a global clock (e.g., the network time protocol[13, 32]).

A basic operation performed by ForceLets is to move a virtual object from one place to another. To perform this task, a combination of acceleration and deccelera-tion phases are employed. There can be many variations for applying these phases, for example:

A **Step ForceLet** applies a constant force onto the virtual object until the midway point, then the ForceLet applies a constant deccelerative force with equal magnitude, after which the ForceLet applies no force:

$$F_f(A, \tau, T; t) = \begin{cases} 0 & \text{for } t - \tau \leq 0 \\ A & \text{for } 0 \leq t - \tau < \frac{1}{2}T \\ -A & \text{for } \frac{1}{2}T \leq t - \tau < T \\ 0 & \text{for } T \leq t - \tau \end{cases} \tag{3.1}$$

A benefit of the step ForceLet is that it is very easy to compute (e.g., an `if` statement). However, it suffers from discontinuities at the extremities and the mid-point. These discontinuities cause a discernible jitter in the object's movement when observed by the user.

The **Half Cosine ForceLet** can be used to smooth the transition from acceler-ation to decceleration. For example:

$$F_f(A,\tau,T;t) = \begin{cases} 0 & \text{for } t-\tau \leq 0 \\ mA\cos\left(\pi\frac{t-\tau}{T}\right) & \text{for } 0 \leq t-\tau < T \\ 0 & \text{for } T \leq t-\tau \end{cases} \tag{3.2}$$

The half cosine ForceLet, however, still has sharp end points. The maximum force is applied at the start and end.

The **Sine ForceLet** uses a full phase of the sinusoidal function which has the advantage of a gentle acceleration and decceleration periods as well as maintaining the smooth transition in the middle. For example:

$$F_f(A,\tau,T;t) = \begin{cases} 0 & \text{for } t-\tau \leq 0 \\ mA\sin\left(2\pi\frac{t-\tau}{T}\right) & \text{for } 0 \leq t-\tau < T \\ 0 & \text{for } T \leq t-\tau \end{cases} \tag{3.3}$$

The simple ForceLets, as described above, work well for moving objects from one point to another in space, but do not work well for moving objects along a complex curve. Applications will often require moving objects along curves through space, not simply moving them directly to the destination.

An example of this is a robot route planning application. In this type of application, the robot is required to follow a complex path (possibly avoiding obstacles) to achieve its goal.

One of the strengths of the ForceLet approach is that we may define the entire path ahead of time and transmit it to the slave objects at once. ForceLets are combined by simply adding them to form a ForceLet group. For example:

$$\mathcal{F}_f(\underline{A},\underline{\tau},\underline{T};t) = \sum_{i=0}^{i=n-1} \mathcal{F}_f(A_i,\tau_i,T_i;t) \tag{3.4}$$

where $\underline{A}$, $\underline{\tau}$, and $\underline{T}$ are n-ary vectors of amplitude, start time and durations.

In addition to ForceLet groups, we also need to assign positions in three-dimensional space. We do this with a vector of ForceLet groups:

$$\langle X, Y, Z\rangle(t) = \langle \mathcal{F}_d(\underline{A}_X, \underline{\tau}_X, \underline{T}_X; t), \mathcal{F}_d(\underline{A}_Y, \underline{\tau}_Y, \underline{T}_Y; t), \mathcal{F}_d(\underline{A}_Z, \underline{\tau}_Z, \underline{T}_Z; t)\rangle \tag{3.5}$$

where the ForceLet group parameters are indexed by a dimension (e.g., X, Y, Z) to show that the dimensions operate independently.

ForceLet group vectors can also be made to operate in orientation dimensions (e.g., Euler angles). NAVL defines the following Euler angles:

θ A counter-clockwise (CCW) rotation about the Z axis

γ A CCW rotation about the X axis.

λ A CCW rotation about the Y axis.

Unlike the Cartesian coordinate system, Euler angles do not commute – the order of applying Euler angles must be specified. In NAVL, we defined the order as θ, γ then λ.

ForceLet Example

Figure 3.4 shows a profile of a simulation of the effects seen as part of the coherency algorithm. The source is a sinusoidal function, and the ForceLet group is made to match the source path precisely. Two other coherency algorithms are shown, the dead reckoning (in velocity) and the stream of data approaches.

Some results are summarized in Table 3.1. Notice that the stream of data approach (a very commonly used approach) has the most updates and the worst error (both max and average), while the ForceLet approach has the lowest and best error (both

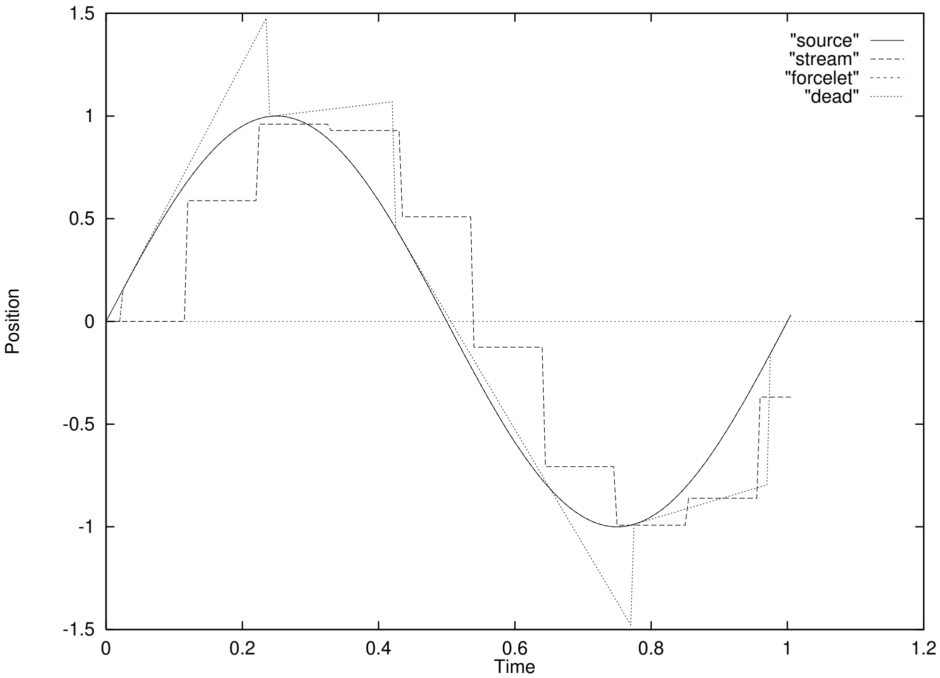


Figure 3.4: Profile for coherency algorithms. The network delay was set at 20 ms, the sample frequency for stream of data was set at 10 Hz, and the threshold for the dead reckoning algorithm is set at 0.5

Table 3.1: Comparison of coherency algorithms

Algorithm	No. of Updates	Max. Error	Avg. Error
Stream	15	0.727	0.258
Dead Reckoning	6	0.609	0.146
ForceLet	1	0.125	0.00155

max and average). The ForceLet approach clearly outperforms the stream of data and dead reckoning coherency algorithms.

While this is a simple example, it demonstrates the strengths of the ForceLet model – i.e., less error and fewer updates.

3.4.3 Route Mapping

A ForceLet groups allow applications to specify a complex curve through space (such as an electron following the lines of force around a magnet). However, applications are likely to use a different representation for curves that need to be translated into the ForceLet group. For example, a curve may be specified by a series of reference points/times or by a reference curve (e.g., a solution to a differential equation).

The task of converting from an internal curve representation to the ForceLet group is called route mapping. Figure 3.5 shows a route mapper capable of translating either a set of reference points or a set of parametric functions into a ForceLet group.

The route mapper can be thought of as either a NAVL service, or an application service. Application services are units of functionality that live in the application domain (i.e., they have intimate knowledge of the application's task, algorithms, or data structures), whereas NAVL services are broad units of functionality that service many application domains.

In addition, application services can be supplied as source-code for inclusion into the application, while NAVL services are provided through an Application Programming Interface (API). Providing a route mapper as an application service allows it to be modified, as needed, to fit the internal curve representation. A NAVL route mapper service would need to support multiple curve representations. At minimum, it should support both representations found in Fig. 3.5.

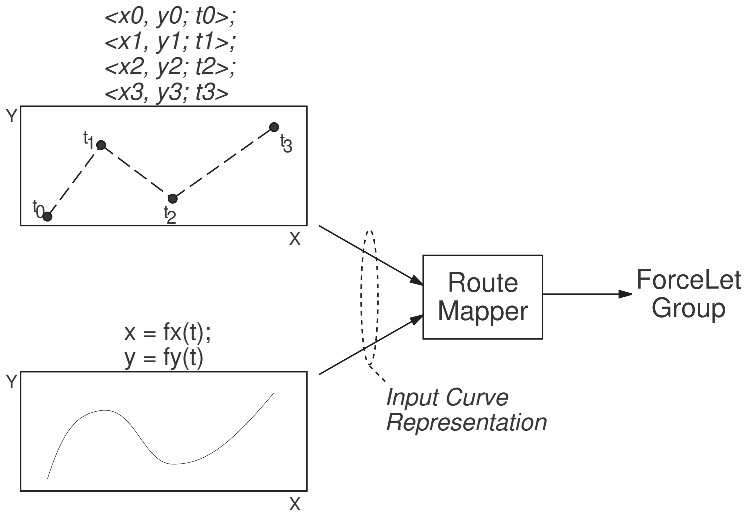


Figure 3.5: Two dimensional route mapping using two input curve representations – a set of reference points and a set of parametric functions

3.5 Rendering and Collision Detection

Visual representation of virtual objects is a critical component to nearly all VR implementations. Rendering is the process of drawing a view into the virtual environment onto a 3D or 2D imaging device such as a head mounted display (HMD) or computer monitor. In most cases it involves a projection from the virtual environment database (defined in so-called world coordinates) onto a display device such as a monitor screen, HMD view plane, or even the walls of your office[20].

Applications need to know when objects in the virtual environment collide. This is used by the system to effect behavior as expected within the scope of the application. Collision detection is the process of determining (or in some cases, predicting) these events.

3.5.1 Rendering

From the architecture point of view, there are two natural places to locate the rendering system: (1) the rendering system can be placed in the NAVL host, or (2) the rendering system can be integrated with the application. Recall from Fig. 3.2 that the application is external to the NAVL host, thus the placement of the rendering system boils down to inside or outside of the NAVL host.

The advantage of placing it inside the NAVL host is that the NAVL system now has the ability to manage their resources – in particular, the memory and CPU resources. The disadvantage is that now the NAVL host is responsible for the entire GUI environment. It is not feasible to separate the 2D GUI graphic devices from the 3D display screen. Application developers often need to be able to integrate 2D and 3D objects into an effective display for many application genres.

In addition to the location of the rendering system, the model of the virtual objects must also be defined. There are two general forms for describing 3D shapes[19].

Surface Modeling is when the skin of the virtual object is modeled. For example, B-spline, NURB, and simple polygon representations are common forms of surface modeling. Surface modeling is popular because it provides a mechanism for drawing complex shapes. For example, articulated figures with integrated joints and wrinkles in cloth.

However, the extra flexibility also involves an extra cost due to complexity. Many VR environments use several models with different Level of Detail (LoD) settings. For example, the human skeleton is represented in [4, p. 125] with a high LoD figure with 131,275 polygons and as a low LoD figure with 8,979. The situation is made worse by the complexity of deciding when to switch between LoD models (e.g., the decision is based on distance, speed of object (optic flow), etc.).

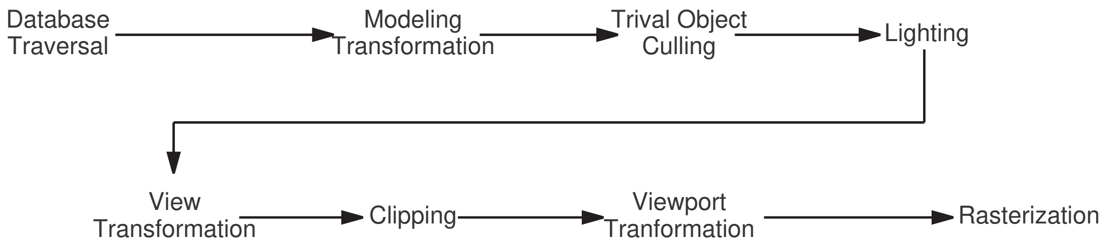


Figure 3.6: The graphics pipeline of [19, Ch. 16, 18].

Solid Modeling is when the object is modeled as a combination of low-level geometric object such as a sphere, cube, or cone. These objects can be combined using Boolean operation (e.g., a cube with a cylinder removed from the center looks like a nut, while a cylinder with a cube on the end looks like a bolt). The solid modeling approach is well suited for CAD applications because they often involve the manipulation and design of solid objects (i.e., the modeling form is very close to the application domain). This modeling approach is also very useful for fast VR systems. The down-side of solid modeling is that they often do not have the richness of a surface modeled object, especially for complex shapes.

The process of rendering the virtual objects is shown Fig. 3.6[19]. The stages of the processing pipeline are:

Database Traversal loops through the environment database to push objects into the pipeline. In the case of solid modeling, the model may have to be converted into a surface model such as a set of polygons.

Modeling Transformation converts the object's coordinates into the world coordinates. Recall that each object has its own coordinate system; these need to be converted in order to compare relative positions of objects.

Trivial Object Culling removes objects that are obviously not within the view region. This is often performed by clipping against a simple viewing box.

Lighting adjusts the pixel or polygon intensity based on the position of the viewer and any lights shining on the surface along with surface parameters (e.g., color, shinyness).

Viewing Transformation converts the world coordinates into normalized projection coordinates (NPC). This transformation performs perspective projections.

Clipping removes any pixels or polygons (depending on the type of clipping) that lie outside the view cone.

Viewport Transformation converts the NPC coordinates into screen coordinates.

Rasterization draws the image into video memory for display. This process often includes a Z-buffer mechanism for depth culling.

Please note that these phases are often combined together. For example, the first three phases (database traversal, object transformation, and trivial culling) can often be put into the same stage.

In addition, special graphics acceleration hardware is used to speed up the throughput of the graphics pipeline. Typically these systems interface with a graphics API such as OpenGL[37], Glide[1], or Direct3D[30].

3.5.2 Collision Detection

Just as with the rendering system, the collision detection system can be placed within the NAVL host, or within the application. Quite likely, the choice will hinge on the design choice made for the rendering system.

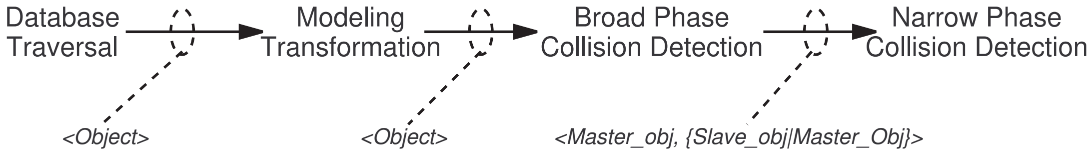


Figure 3.7: The collision detection pipeline adapted from [33].

The process of detecting collisions generally follows a pipeline similar to the rendering pipeline (see Fig. 3.7). The stages are defined as follows:

Database Traversal loops through the environment database.

Modeling Transformation converts the object's coordinates into world coordinates.

Broad Phase Collision Detection generates a cartesian product of the master objects with the slave objects and filters out objects that are clearly not colliding. In NAVL, only master object collisions need to be examined. Slave-to-slave collisions may be ignored because there will always be a master object on a NAVL host that will observe the collision on its local host. This reduces the possible number of candidate object pairs considerably. To reduce it further, a bounding box collision detection technique may be employed[33].

Narrow Phase Collision Detection reduces the set of candidate object pairs to the final set of collision events. This requires a sophisticated object collision detection algorithm, such as that seen in [27, 24, 33]. These techniques all require extensive knowledge of the object model (e.g., polynomials and their normals).

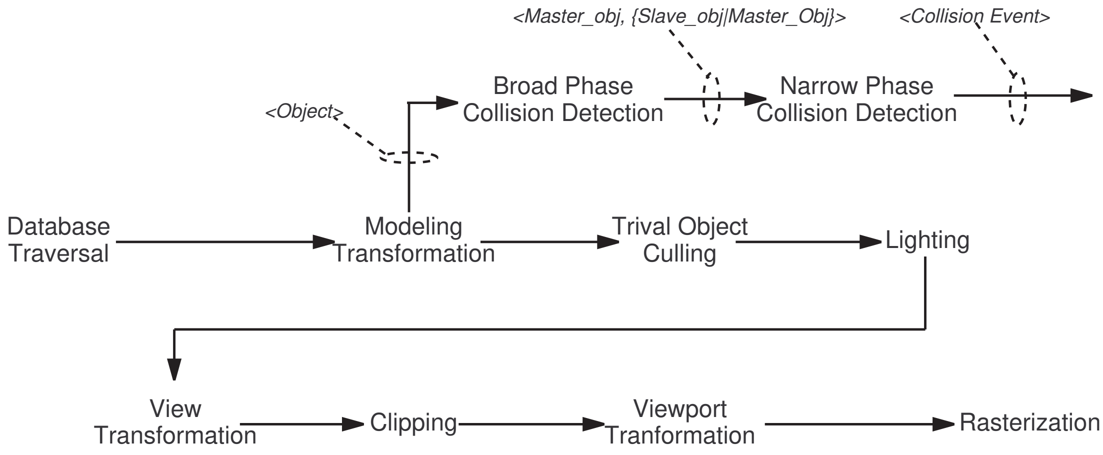


Figure 3.8: The combined rendering and collision detection pipeline

3.5.3 System Perspective

One can see from Figs. 3.6 and 3.7 that there is considerable overlap of much of the graphics and collision detection pipelines. Figure 3.8 shows a combined pipeline.

The benefits of such an arrangement is the code reuse and improved utilization of system resources. However, the disadvantage of combining the two pipelines is that both pipelines must be run at the same rate. Often it is desirable to run the rendering pipeline at a constant rate determining the frame-rate (e.g., 20 Hz-60 Hz) while the collision detection pipeline will run at a rate appropriate to the remaining CPU time.

An additional simularity between these systems is that both rendering and collision detection benefit from exploiting coherence. In this context, coherence is the property that the virtual objects move only small distances between frames, and should not be confused with coherency as used for master/slave object synchronization.

For rendering, coherence means that objects are not visually changing between frames. This allows the rendering system to simply re-rasterize the previous image[19] rather than perform a complete transformation and projection.

For collision detection, coherence allows the collision detection mechanism to disregard object pairs that are known not to be within collision distance[27, 24]

3.6 Chapter Summary

This chapter introduces the NAVL system architecture. The architecture is made up of three components:

1. Network Architecture
2. Object Distribution and Coherency Model
3. Rendering and Collision Detection

The **network architecture** uses a distributed client/server topology with broadcast LANs and peer-to-peer WAN interconnect. This spreads the load between the participants while still reducing the flooding effects noticed with pure peer-to-peer topologies.

A special feature of the network architecture is the message manager which acts as a gateway between the LAN and the WAN, a filtering mechanism to prevent unwanted packets from leaking into the WAN (or LAN), a directory service to locate virtual objects and users, and a QoS control mechanism to regulate the QoS parameters over the WAN interconnect.

The **object distribution and coherency model** describes autonomous objects with a master/slave distribution mechanism. The master updates slave objects with ForceLets.

ForceLets are messages that describe the application of a force to the object with a synchronized clock. This mechanism allows for ForceLets to push objects along a specified path. Three ForceLets were examined: the step ForceLet, the half cosine ForceLet and the sine ForceLet. The sine ForceLet provides the smoothest transition from acceleration to decceleration.

The function of a route mapper system was also presented whose purpose is to map application domain curve representations (e.g., reference points or curves) into Forcelets.

The **rendering and collision detection** systems were also presented in turn via an examination of their pipelines. After looking at the respective pipelines, it was determined that there is much in common between these systems; a combined pipeline was proposed.

Chapter 4

NAVL Prototype

A prototype of the NAVL architecture has been developed as a proof-of-concept and test-bed for evaluating the NAVL system architecture. For these purposes, a full implementation of the NAVL architecture was not required. Therefore, the prototype exhibits the following limitations:

- **Minimal Message Manager Support**: A limited message manager is included that supports clock synchronization and a simple name service, but not QoS, Filtering, or Gateway services. Without gateway services, the NAVL prototype is limited to a single LAN.
- **UNIX–Only**: The prototype has been developed using the UNIX operating system; other operating systems would require porting.
- **Single–Threaded**: The prototype implementation is achieved through an internal task dispatcher. Tasks are self–contained and fully executed between task switches, thus no threading support is required (e.g., such as stack and register switching). However, in symmetric multiprocessing environments, a true multi–threading environment would be instrumental to improved performance.

- **Single Application per NAVL Host**: The communications protocol can only support a single application per NAVL host. This was found to be sufficient for our purposes, however a full deployment would extend the capability of the NAVL host to include multiple applications.

This chapter shows the design choices made during the implementation of the prototype. The implementation architecture describes *how* a design is constructed in response to a system architecture such as Ch. 3. The overall implementation architecture is discussed first, then followed up by details of the specific components: object simulation and execution unit, rendering and collision detection, and finally the network subsystem and protocols.

In addition to the design elements, this chapter also describes the Application Programming Interface (API) and Object Programming Interface (OPI). These interfaces help separate the domains of the application programer from the object programer. This makes it easier for either the application programer or the virtual object programer as each person does not require knowledge of the other.

A route mapping system is also introduced as a candidate method to create groups of ForceLets to follow pre-determined paths (specified by reference points).

4.1 Overview

The NAVL prototype implements services with an extended DVR layer as shown in Fig. 4.1. The top layer is separated into an application layer and a master object layer. This emphasizes the architectural choice involved with having autonomous objects. The autonomous objects have execution code within themselves that are accommodated through the master object layer on down.

<table>
<tr><td>(Layer 1)</td><td colspan="2">Application</td><td></td><td>Master Object</td></tr>
<tr><td>(Layer 2)</td><td>Rendering API</td><td>NAVL API</td><td></td><td>NAVL OPI</td></tr>
<tr><td>(Layer 3)</td><td>Renderer</td><td colspan="3">Programming System</td></tr>
<tr><td>(Layer 4)</td><td>Window System</td><td colspan="3">Operating System</td></tr>
<tr><td>(Layer 5)</td><td>Window System Protocols</td><td colspan="3">Protocols</td></tr>
</table>

Figure 4.1: The service layer diagram for the NAVL prototype

On the surface, this would seem to increase the complexity for programming applications for the NAVL prototype as compared to a simpler layering structure. However, the master object's execution code may be thought of as tool kits for constructing the active portions[1] of a VR application

Toolkits are commonly used by venders to encapsulate units of extensibility. For example, the X-ToolKit is a library that allows X-Window developers to create GUIs. In a commercial deployment, a NAVL vender would typically provide one or more toolkits for application developers to use when building their applications.

The master objects have their own programming interface to some of the same routines available within the programming layer (layer 3). This organization improves utilization of the code base due to the large overlap between the application and object services.

The block diagram of the NAVL prototype is shown in Fig. 4.2. The heart of the system is the ForceLet simulation and behavior execution unit (SEU). It is responsible for executing behavior routines found in the master objects as well as simulating ForceLets.

The network engine provides protocol encapsulation and transport services to the

[1]The active portions are the autonomous object's behavior routines. Passive elements include the object's shape and color.

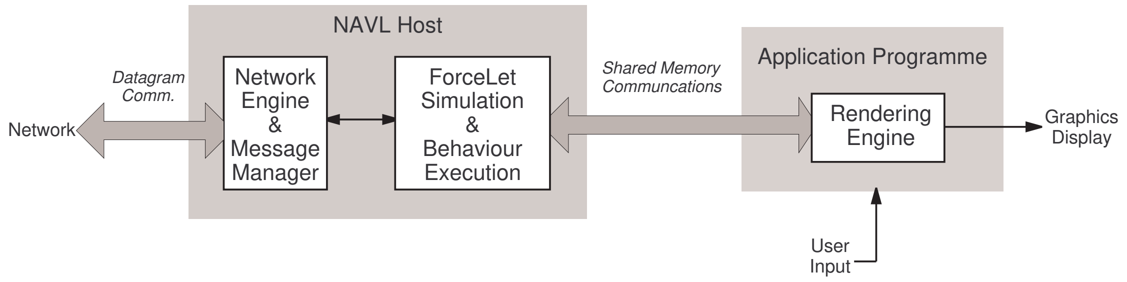


Figure 4.2: Block diagram of the NAVL prototype

NAVL host. This includes the transmission of ForceLets, state changes, and other utility functions.

The message manager has a very limited role in the prototype due to the single-LAN implementation, the message manager is responsible for ensuring that all NAVL hosts have a synchronized clock and provides some name service mapping between published object names and their object IDs. In a full implementation, the message manager would also be responsible for QoS, Filtering and Gateway services.

In the prototype, the code for the message manager is linked into every NAVL host. According to the NAVL system architecture, only one message manager is needed per LAN, therefore only the message manager linked to NAVL host number one is executed.

The rendering engine is supplied as a library that is linked directly into the application program (along with the API services). The rendering engine is responsible for drawing the scene. The collision detection services are currently located in the ForceLet simulation and behavior execution unit. However, because there is much overlap between the rendering and collision detection operations, the collision detection mechanism is discussed in the same section as the rendering engine[2].

Shared memory is used to communicate between the application and the NAVL host. It has a low overhead and is well suited for high bandwidth communication tasks (such as rendering). A broadcast datagram protocol is used between NAVL hosts. Datagrams have lower overhead compared to virtual circuits.

4.1.1 Service Dispatcher

The prototype implementation components of Fig. 4.2 are scheduled using a task dispatcher (not shown in the figure). The dispatcher is essentially an endless loop of service request cycles; each cycle performs the steps shown in Fig. 4.3. The cycle rate is regulated at 20 Hz to ensure a consistent environment lag for the rendering services, thus presenting a stable refresh rate for the user.

The order of tasks is chosen to enable a waterfall model of data flow; for example, collisions are detected prior to handling events to enable quick collision handling. The more important tasks are arranged at the top of the list to establish a priority queue. In future implementations, a preemptive scheduler can be employed to allow long running tasks that are interrupted by higher priority tasks when needed.

4.2 The Object Simulation and Execution Unit

The object simulation and execution unit (SEU), shown in Fig. 4.4, provides the bulk of the object and application services found in layers 2 and 3 of the DVR layer stack

[2]Due to the degree of existing support for rendering services in the OpenGL library, it was decided that the prototype would not use a combined collision detection and graphics pipeline – the amount of rework is too high given the expected benefits.

Task	Description
Get Start Time	Used to regulate the cycle period (see Sleep step).
Simulate Objects	Executes ForceLets for each slave object in the object database. The results are placed in the object state table.
Detect Collisions	Loop through the object state table and detect collisions between master objects and slave or other master objects. When collisions are detected, a collision event is issued to the master object for both objects involved.
Run Events	Execute behavior routines for objects with pending events.
Poll API	Read the API communication buffer for pending API requests. The requests are handled immediately.
Poll Console	Poll the keyboard of the NAVL host to service operator requests.
Poll Network	Read the network communication buffer for pending network requests including requests for the message manager. Any pending requests are handled immediately.
Sleep	The duration of the service cycle is determined from the start and end times. To achieve a specified cycle rate (default is 20 Hz), the NAVL host sleeps for the remaining time, if any.

Figure 4.3: Steps taken during a dispatch cycle

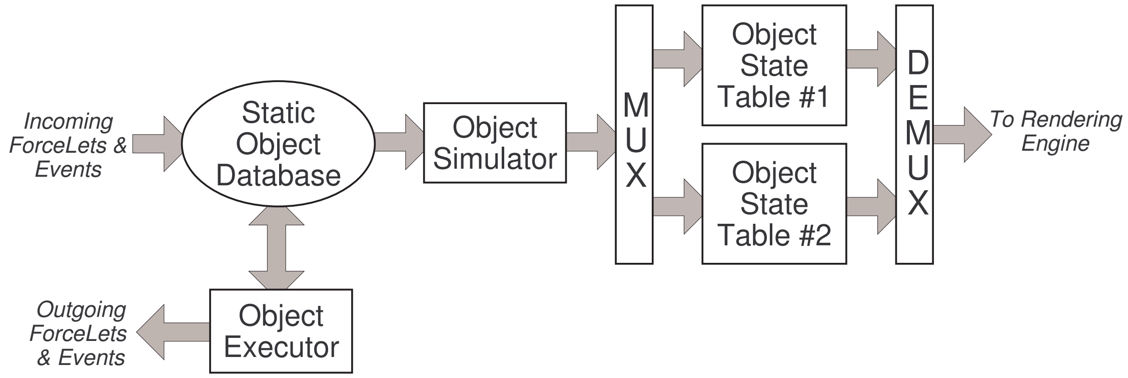


Figure 4.4: The virtual object simulation and execution unit (SEU)

(i.e., all except for rendering services). These services include: (1) Object simulation (e.g., ForceLets), (2) Object behavior execution, and (3) API and OPI.

4.2.1 Object Simulation

Upon request by the service dispatcher, the object simulator executes an object's ForceLets and stores the result into one of the two state tables. A double-buffering mechanism is used on the state tables to allow the SEU to update one of the state tables while the rendering engine reads from the other. The state tables are placed in shared memory for quick access by both the object simulator and the rendering engine.

Static information, such as object shape and color, remains in the static object database and is made available to the rendering engine through the API.

The objects are placed in a hierarchy to allow for easy manipulation of groups of objects similar to [19]. For example, Fig. 4.5 shows an articulated mannequin realized using a hierarchy of torso, head, arms, legs, hands and feet.

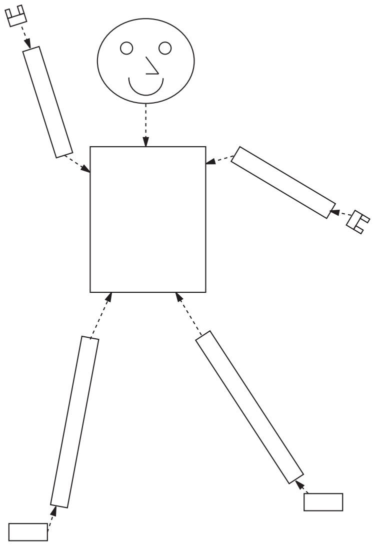


Figure 4.5: Object hierarchy when making an articulated mannequin. The arrows indicate child to parent relationships

The hierarchy reduces the amount of CPU power expended during ForceLet simulation. For example, to make the mannequin walk without object hierarchies requires moving all the objects independently (at least ten cartesean ForecLets, and about eight orientation ForceLets). However, if object hierarchies are used we can move parent objects and let the child objects move with the parent (reducing the number of ForceLets to about one cartesean ForceLet and four orientation ForceLets).

4.2.2 Object Behavior Execution

The object behavior executor obtains behavior routines from the static object database and executes them upon request by the service dispatcher.

To promote reusability, object's behavior code can be shared between many instances of the object. For example, there can be several copies of the mannequin.

Each copy in the NAVL host has its own context, but share the same behavior routines.

For nomenclature purposes the collection of shared behavior routines is called an *object class* and the context surrounding a specific master object is called an *object instance*. The terms "object instance" and "master object" are interchangible.

The object class is implemented as C–language routines within a shared library. When the master object is created (at the request of an application), the NAVL host determines if the object class has previously been loaded and loads it if needed.

Since the behavior routines are reused between object instances, they must be re–entrant – that is, they must not save state between executions except in designated places. For example, global variables are not allowed because it is unknown which object instance will execute the behavior routine next.

Behavior routines maintain state between invocation in a special instance memory block that is passed into the behavior routine by the NAVL host (i.e., the programming system layer). The NAVL host is responsible for managing and swapping these instance memory pointers as needed.

The master object executes a special startup behavior routine called `initObj` when it is created. This routine creates data structures used by the master object (i.e., the instance memory), and registers the behavior routines. Removal of the master object (at the request of an application) executes a clean-up routine called `freeObj` to free any data structures that may have been allocated during initialization.

4.2.3 Application and Object Programming Interfaces

The application and master objects make requests of the NAVL programming system layer (recall the NAVL service layers of Fig. 4.1) through an interface called the

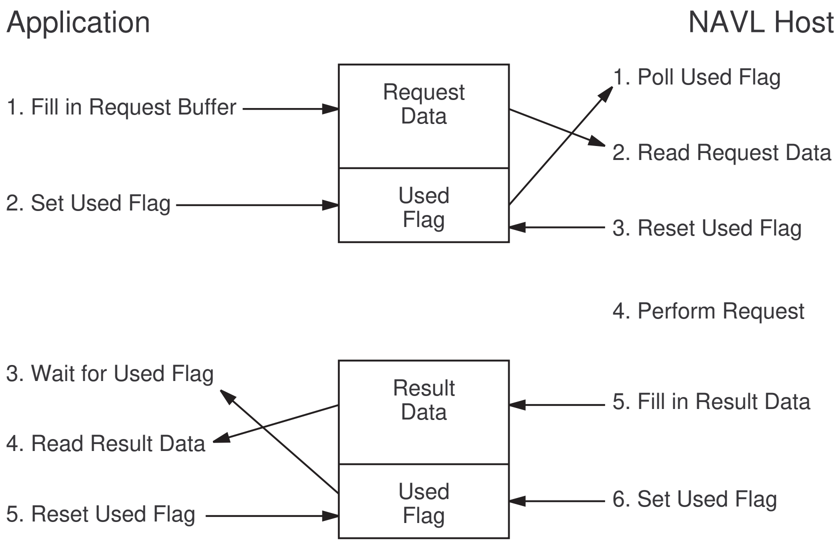


Figure 4.6: API protocol

application programming interface (API) and object programming interface (OPI).

The API forwards requests to the server using a shared memory communication buffer. Shared memory is preferable over network (e.g., IP) interfaces in that it is much more efficient. Network interfaces require system calls to poll the network layer of the OS, while shared memory polling is a simple matter of examining a memory location. A down-side of using a shared memory protocol is that the application must live on the same node as the NAVL host.

A communication protocol has been developed to coordinate between the client and the NAVL host. The request buffer is a simple data containing the command and its associated parameters. A *used* flag synchronizes the read and write operations on the buffer as seen in Fig. 4.6.

A MUTEX is needed to support multiple clients communicating over the same shared memory buffer. In the prototype, however, we allow only a single application per NAVL host, so the MUTEX is not required.

The API requests that may be made are shown in Table 4.1. In general, the API is segmented into a class of requests made to the NAVL host (e.g., `getMode`), requests made to an object (e.g., `sendMsg`) and requests made to the message manager (e.g., `locateObject`). Additional API calls provide debugging and benchmarking support (e.g., `moveObject`).

An example of the API is provided in Fig. 4.7. The application performs as follows: if it is hosted by NAVL host 1, it creates an object called `testobj` (class is `test`) and attaches to the `targetobj` object. Otherwise, it creates the `targetobj` object (class is `target`) and attaches to the `testobj` object.

This particular example was used to test the API's ability to create and attach to objects. The application is started up on two NAVL hosts and they create and attach to each other's object.

The OPI uses a direct invocation call mechanism – the OPI is a wrapper to the NAVL host service. In this way, the object's behavior routines execute from *within* the NAVL host.

This is done for two reasons: first, it is much more efficient to directly invoke NAVL services than to do a polling mechanism as done with the API; and second, the object behavior code is expected to be more robust than applications and thus can be trusted to call NAVL services directly.

The set of OPI calls is shown in Table 4.2, broken into categories for the active and passive elements (and a miscellaneous).

An example of an object class is shown in Fig. 4.8. This example uses a class library to create a white sphere initially starting at (10, 10, 0) and moving towards the upper right corner when viewed along the Z–axis. The `initObj` and `freeObj`

Table 4.1: API Calls. The (R) symbol indicates a return parameter

(a)**NAVL Host**

Name	Parameters	Description
connectNAVL	HostID UserId(R)	Connects to the NAVL host specified by hostID, and returns the user id
disconnectNAVL	HostID	Disconnects to the NAVL host.
getObjState	ObjState(R) NumStates(R)	Gets the current object state buffer (NumStates returns the buffer size).
freeObjState	—	Releases the object state buffer.
getMode	Mode(R)	Get the NAVL host's mode. Values can be: INIT, OK, EXITING, or EXITED.
getUID	ID(R)	Get the current user's ID.
getTime	Stack(R)	Returns current NAVL time.

(b)**Object**

Name	Parameters	Description
newObject	Id(R) Name InitData InitSize RefID	Creates a new object (i.e., master) local to the application's NAVL host. The object's identifier is returned in ID.
attachObject	ID	Creates a slave object attached to the specified object ID.
freeObject	ID	Releases a master or slave object.
sendMsg	ID Msg MsgSize	Sends a message to an object.

(c)**Message Manager**

Name	Parameters	Description
publishObject	ID Name	Registers an object with the message manager.
retractObject	Name	Releases the message manager registration.
locateObject	ID(R) Name	Get the ID of a registered object.

(d)**Debugging**

Name	Parameters	Description
moveObject	ID Pos	Moves an object. Normally an application makes a request to the object to move itself; this is used for testing

```
void CreateScene()
{
    /*
     * If we're host 1, then we'll create the cube test object
     * otherwise, we'll attach to it.
     */

    if (host == 1)
        {
        newObject(&id, "test", NULL, 0, ether_oid);
        publishObject(id, "testobj");

        while (locateObject("targetobj", &id2) != RC_OK)
            usleep(100000);
        attachObject(id2);

        newText(&id3, id, "NAVL HOST 1", 5.0, red,
            base_pos, base_orient);
        publishObject(id3, "testtext");
        }
    else
        {
        while (locateObject("testobj", &id) != RC_OK)
            usleep(100000);
        attachObject(id);

        newObject(&id2, "target", NULL, 0, ether_oid);
        publishObject(id2, "targetobj");

        while (locateObject("testtext", &id3) != RC_OK)
            usleep(100000);
        attachObject(id3);
        }
}
```

Figure 4.7: Example of application code

Table 4.2: OPI Calls. All OPI Calls return a status code through the stack. Parameters marked with (R) indicate return parameters; unmarked parameters are input parameters

(a)**Active Elements (Behavior Routines)**

Name	Parameters	Description
OPIsetTimer	ID Ts Period Cb	Creates a timer to execute a callback
OPIsetMsgCB	Id Cb	Establishes a message callback.
OPIsetCollCB	Id Cb	Establishes a collision detection callback.
OPIrmCB	Id EventType Cb	Releases a callback.

(b)**Passive Elements (Object Attributes)**

Name	Paramaters	Description
OPIissueFL	Id FlType Dimension T0 T A	Issues a ForceLet to all slave objects.
OPIsetRadius	Id Radius	Sets the radius of the object.
OPIsetShape	Id Shape	Sets the shape of the object.
OPIsetOrient	Id Orient	Sets the orientation of object.
OPIsetPos	Id Pos	Sets the position of object.

(c)**Miscellaneous**

Name	Paramaters	Description
OPInewObject	RetObj(R) Class InitData InitSize RefId	Creates a new object.

functions are required as part of the class library to initialize and clean-up the master object.

4.3 Rendering and Collision Detection

These two components are discussed together because there is much overlap between their pipelines as seen in Sec. 3.5. However, given that graphics rendering libraries already exist, it was decided to actually implement these components independently. The rendering service is implemented using OpenGL and the collision detection service is implemented with a nested-loop mechanism.

4.3.1 OpenGL–Based Rendering

The NAVL prototype renders scenes obtained from the NAVL host's state table. The OpenGL graphics library is used for drawing objects and the `GetObjState()` NAVL API call is used for obtaining the state table.

The OpenGL graphics library simplifies many of the rendering tasks, such as:

1. Drawing tasks (e.g., perspective projections, hidden surface removal, object rendering – sphere, cube, etc.).
2. Shape transformation (e.g., rotations and translations).
3. Object hierarchies (e.g., through nested object matrices).

In addition, the OpenGL graphics library is widely available in both software and hardware–accelerated forms[34, 37, 46]. This improves the portability of the

```
void *initObj(Id_t oid, char *init_data, int init_size)
{
    Coord_t   pos    = {10.0, 10.0, 0.0};
    Orient_t  orient = {0.0, 0.0, 0.0};
    Shape_t   shape;

    shape.type             = SHAPE_SPHERE;
    shape.is_solid         = TRUE;
    shape.colour           = white;
    shape.data.sphere.size = 5.0;

    OPIsetShape(oid, shape);    /* Set shape to white sphere     */
    OPIsetOrient(oid, orient);  /* Set orientation toward viewer */
    OPIsetPos(oid, pos);        /* Set position to 10,10,0       */

    /* Move the sphere at 45 degree angle for 10 seconds */

    OPIissueFL(oid, sine, X, getTimeSec(), 10.0, 2.0);
    OPIissueFL(oid, sine, Y, getTimeSec(), 10.0, 2.0);

    return NULL;
}

void freeObj(void *class_data)
{
    printf("Object termination\n");
}
```

Figure 4.8: Example of object class behavior routines

Name	Paramaters	Description
changeColour	Param Colour	Changes the current drawing color model: ambient, diffuse, specular, emission, and shininess.
drawText	Font Size FmtString	Draw an ASCII text string.
drawShape	Shape	Draw a shape (e.g., sphere or cube).
drawShapes	ObjState	Draw the shapes in the object state buffer.
drawFrame	—	Draw a frame (get the object state buffer and draw it).

Table 4.3: Client Helper API Calls. These calls wrap around existing NAVL API and OpenGL API calls to make it easier for the application developer to draw NAVL shapes

NAVL prototype, while also providing an easy upgrade path to a faster graphics implementation.

To simplify the application programming task, a client rendering core object class were created. The helper library wraps around the OpenGL graphics library providing routines for drawing NAVL objects.

The core object class provides a minimal toolkit for applications to create base-level objects (such as spheres or cubes). The toolkit is composed of the object class and an interface module for applications. The header files for both the client library and core API library are shown in Appendices B.5 and B.6. The client API is also given in Table 4.3.

The collection of shapes available in the prototype is shown in Table 4.4. The origin shape has no display, but can be used as a parent shape to link two sibling shapes in the same object hierarchy. For example, the case of two spheres circling each other might be best represented as an origin object (located in the center) with the two spheres as child objects. The origin shape creates a local coordinate system

Shape	Shape Parameters	Description
origin	—	Used for specifying object hierarchies
sphere	Radius	Regular Sphere
cube	Size	Regular cube
cone	BaseSize, Height	Regular closed tip cone
torus	InnerRadius, outerRadius	Doughnut
teapot	Size	Traditional Teapot
text	Font, Size	Latin–1 Locale Text

Table 4.4: Core shapes of the NAVL prototype. All shapes can specify its color using an RGBA color model and may also specify whether the shape will be drawn with a solid or wireform rendering model

for the spheres to be placed into.

4.3.2 Nested–Loop Collision Detection

For simplicity (see Fig. 4.9), the NAVL prototype uses a nested-loop collision detection algorithm. This simplicity, however, leads to efficiency problems (i.e., quadratic algorithmic complexity). This is tolerated in the prototype due to the small number of objects. Section 3.5.2 describes an idea for improving the efficiency of the collision detection mechanism by leveraging the common pipeline between these two services; however, this was not implemented as part of the prototype.

Issuing multiple collision events for the same collision can be a major problem for object developers. The algorithm as given here will emit a collision event each time it is detected between two objects regardless of whether it is detecting the same collision multiple times.

This is alright if the collision detection routine handles the event in such a way as to prevent a further collision such as moving the object out of the way. However, sometimes the developer doesn't want to move the object. Perhaps the object should

```
for (o1 = root_obj; o1 != NULL; o1 = o1->next)
    {
    if (isCDEnabled(o1))
        {
        for (o2 = o1->next; o2 != NULL; o2 = o2->next)
            {
            if (isCDEnabled(o2)
                {
                d_min = o1->radius + o2->radius;
                if ((dist(o1, o2) <= d_min) &&
                   (isMaster(o1) || isMaster(o2)))
                    SendCollisionEvent(o1, o2)
                }
            }
        }
    }
```

Figure 4.9: Code snippet of the nested–loop collision detection routine

only glow red when in a collision state with another object. In this case, the algorithm will produce many collision events.

To handle this, a collision detection "debouncing" mechanism can be used as follows: when the collision is first detected, the object stores a record of the event and a timestamp. Upon the next collision, the collision detection routine checks to see if the same collision occurred; if so, the timestamp is updated and the collision detection routine is not called, otherwise the collision is handled normally.

The debouncing mechanism was not implemented in the NAVL prototype, however it is still possible to provide these services at the object level, in which case the collision detection behavior code within the master object maintains state regarding the last collision encountered and handles the event only if it is determined to be new.

4.4 Network Subsystem and Protocols

Each host in the NAVL system communicates to each other and their message manager via the network subsystem. The NAVL prototype implements this subsystem using broadcast datagram service[45] supplied by the UNIX operating system through the BSD socket API.

The message manager services are logically orthogonal to the NAVL host; it can reside anywhere on the LAN, so long as there is only one active message manager per LAN. However, for ease of implementation, the message manager is physically connected with NAVL host 1.

The NAVL prototype implements LAN-only network services; thus, the message manager is responsible for two functions: (1) NAVL host clock synchronization, and (2) virtual object directory service. The clock synchronization is used when a NAVL host first announces itself to the LAN upon startup. The name service provides a means to publish, locate, and retract virtual objects as seen in Table 4.1.

4.4.1 UDP Transport Mechanism

The user datagram protocol (UDP) requires both a node address and a port address. The port differentiates between UDP packets destined for the same node. Typically, a separate port will be used for different applications[3].

The broadcast addressing mode enables a NAVL host to transmit a single message to all NAVL hosts on the LAN. This dramatically reduces the number of messages that

[3] The NAVL prototype currently uses port 1515 as it is unused on the development machines — it can be modified if needed. All NAVL hosts on the LAN must communicate on the same port number.

need to be sent. However, there can be a cost involved in broadcast addressing[7, 10], discussed below.

Broadcast packets can affect the performance of machines on the LAN in two ways. First, the transmission of the packet uses up networking resources on the LAN itself. This is an expected cost, and one that is easily accommodated.

Second, due to the segmentation of the hardware and OS drivers, the operating system relies on the network interface card (NIC) to only interrupt the OS when a packet is detected with the node's IP address. In the case of broadcast UDPs, the node's OS will always be interrupted; even if this is not a NAVL host servicing the UDP's port.

Thus every machine on the LAN will be affected by both the diminished network resources and by the CPU time required to read and discard the broadcast packets.

One solution to this problem is to determine the trade-off point where a serialized unicast IP (i.e., sending messages directly to IP addresses instead of broadcasting) has a lesser effect on the LAN than broadcasting. The hosts should use serialized unicast IP whenever there is fewer than this threshold number of hosts in the LAN, otherwise use broadcast.

4.4.2 Protocol Data Unit

The NAVL prototype communicates with structured packets called a protocol data unit (PDU). The following is a C–language representation of the data structures of the PDUs; more details are available in Appendix B.4:

PDU Definition

This is the top–level data structure of the PDU. The `to` and `from` fields define the NAVL host numbers of the sender and receiver. NAVL host numbers are configured by the user, but must be greater than 0 and less than 65535[4].

Note that the NAVL prototype was developed on the Sun SPARCstation running Solaris 2.6, therefore no attempt was made to ensure endian order on integer types. This is a fairly simple task to fix if the need arises.

Field	Description
`int to`	Destination host
`int from`	Sending Host
`NETtype_t type`	Packet Type
`NETdata_t data`	Packet Data

`NETpacket_t` Structure

The following is the definition of the `NETtype_t` enumerated type used in the NAVL PDU. In the C–language, enumerated types start with the integer value zero and increment from there.

[4]The address `0xFFFF` is reserved for broadcasting to all NAVL hosts, and address `0` specifies the message manager.

Name	Description
`NET_REQ_OBJECT`	Request a slave object for attaching
`NET_OBJECT`	Slave object (used for attaching)
`NET_POS`	Object Position
`NET_ORIENT`	Object Orientation
`NET_RADIUS`	Object Radius
`NET_SHAPE`	Object Shape
`NET_FORCELET`	ForceLet sent to Slaves
`NET_MESSAGE`	Message Event
`NET_COLLISION`	Collision Event
`NET_REQ_START`	MM: Request start time for new host
`NET_START`	MM: Start Time Response
`NET_DEF_OID`	MM: Define (publish) Virtual Object
`NET_RM_OID`	MM: Remove (retract) Virtual Object
`NET_REQ_OID`	MM: Get Published Object's OID
`NET_OID`	MM: Published Object's OID Response

`NETtype_t` Enumerated Type

The `NETdata_t` type is a union; that is only one of the fields may be used at a time. The `type` field of `NETpacket_t` (defined above) declares which field element is active.

Field	Description
`NETpos_t pos`	Position
`NETorient_t orient`	Orientation
`NETradius_t radius`	Set Radius
`NETshape_t shape`	Shape
`NETobj_t obj`	Object (used when attaching)
`NETfl_t fl`	ForceLet
`NETmsg_t msg`	Message
`NETcoll_t coll`	Collision
`Time_t start`	Start Time
`NETdef_t def_obj`	Define Object
`char obj_name[MAX_IDENT]`	Object Name
`Id_t oid`	Object ID

`NETdata_t` Structure

API/OPI Call	PDU Sent	PDU(s) Read
attachObject	NET_REQ_OBJECT	NET_OBJECT, NET_POS, NET_ORIENT, NET_RADIUS, NET_SHAPE, NET_FORCELET(s)
moveObject	NET_POS	*none*
sendMsg	NET_MESSAGE	*none*
publishObject	NET_DEF_OID	*none*
retractObject	NET_RM_OID	*none*
locateObject	NET_REQ_OID	NET_OID
OPIissueFL	NET_FORCELET	*none*
OPIsetRadius	NET_RADIUS	*none*
OPIsetShape	NET_SHAPE	*none*
OPIsetOrient	NET_ORIENT	*none*
OPIsetPos	NET_POS	*none*
Host Init	NET_REQ_START	NET_START
Collision	NET_COLLISION	*none*

Table 4.5: API/OPI to PDU mapping

4.4.3 API/OPI to Protocol Mapping

The API and OPI calls from Tables 4.1 and 4.2 define the language–level access to NAVL services. These service requests are translated into a series of one or more network request/response packets. This process is loosely called the API/OPI to protocol mapping.

The mapping made as part of the NAVL prototype is shown in Table 4.5.

4.5 Route Mapping

The ForceLet approach for moving objects through space shows much promise for minimizing the number of network messages required to define the curve; however, it can be difficult to determine the correct set of ForceLet parameters to produce a desired curve. Often, an application will have an internal representation of the curves, such as a series of reference points or spline equations.

In order to leverage the network friendly aspects of the ForceLet approach, the application's curves need to be mapped into a set of ForceLets. This task is called route mapping.

Because the application's curve representation is often specific to the application itself, the NAVL DVR system does not provide a route mapper internally. Instead, an example of a route mapper is presented here that may be used or adapted by an application developer.

For this example route mapper, the target curve is defined by a set of reference points in space and time (i.e., X, Y, Z, and t). The route mapper finds a set of ForceLets that map to a smooth curve that intersects each of the reference points.

Without loss of generality, this route mapper handles a single dimension. Multiple dimensions are built up with successive calls to the route mapper services for each dimension.

The method of solution is a non-linear curve fit technique where we define a cost function that defines the mapping. The cost function is minimized using one of several approaches[17, 35].

A sum-of-squares cost function is used as follows:

$$C = \sum_{j=0}^{N_r-1} \left(\left[x_j - \sum_{i=0}^{N_f-1} F_d(A_i, \tau_i, T_i; t) \right]^2 \right) \tag{4.1}$$

where N_r is the number of reference points and N_f is the number of ForceLets.

Note that the ForceLet is defined in the *displacement* domain, this is because the reference points are defined in the displacement domain.

While Levenberg–Marquardt[35] technique is often used to minimize general purpose cost functions, it requires constructing a Hessian matrix. The conjugate gradient (CG)[17] optimization method is therefore a very attractive alternative.

An initial attempt was to curve fit all $3N_r$ parameters (i.e., each ForceLet has 3 parameters: A_i, τ_i, and T_i). This resulted in poor convergence of the algorithm due to the large plateaus surrounding the active parts of the ForceLet where many of the parameters disappear(see Eqs. 3.1–3.3 as $t < \tau$ and $t > \tau + T$).

The next attempt uses a two-stage place and improve algorithm (the C code is provided in Appendix A. The placement algorithm determines the number of and location for the ForceLets (e.g., τ and T), while the improvement algorithm uses a CG optimization algorithm with the position parameters fixed to determine the amplitude (A_i).

The placement algorithm locates a ForceLet between each pair of reference points (in terms of time) with a specified overlap (a parameter of the optimization step, represented as a percentage).

A pruning stage removes ForceLets that contribute little to the path of the virtual object by eliminating ForceLets whose amplitude falls below a threshold (currently set at 0.1).

An example of the route mapper is given in Fig. 4.10. The reference points given in the example are: (x=0, y=0; t=0.0), (0, 5; 1), (5, 5; 2), (6, 0; 3), (10, 3; 4) and the resulting Sine ForceLet parameters are shown in Table 4.6. The dots in the figure are specified 0.1 seconds apart; those that are closer together indicate a slower velocity,

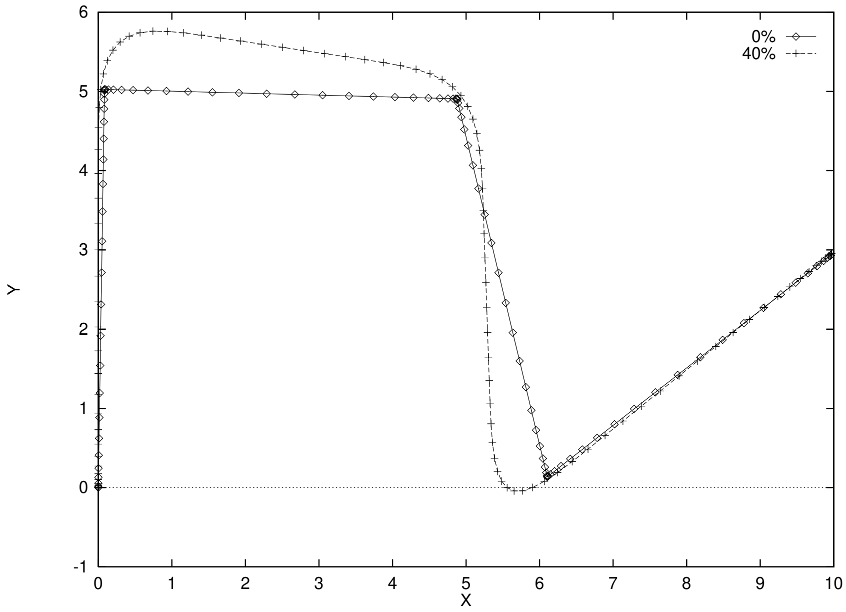


Figure 4.10: Example of route mapping

while dots further apart indicate faster velocity. The route mapper was run with an overlap of 0% and 40%. The 0% overlap curve shows a jerky curve with many starts and stops, while the 40% overlap shows a fluid movement from the start to the end.

4.6 Project Planning

The NAVL prototype was implemented using standard techniques of the software life-cycle[41]. The main tasks are:

Functional Specifications: This is the task of writing down the purpose and function of the NAVL system – what problems it solves, and some details of the proposed solution. The documentation resulting from this task is contained within the oral proposal given to the dissertation committee and a conference paper[51, 56].

Dimension	A	τ	T
X	15.922	0.857	1.428
X	0.655	1.714	1.428
X	14.119	2.571	1.428
Y	18.183	0.000	1.428
Y	-2.217	0.857	1.428
Y	-17.398	1.714	1.428
Y	10.536	2.571	1.428

Table 4.6: Sine ForceLets parameters used as part of the route mapping example

Design Specification: The design specification provides specific implementation details regarding the product – more an issue of how the product works rather than what it does. Along with the functional specification, the design specification formed the basis for the oral proposal and conference papers[51, 56, 57].

Coding and Unit Testing: The coding task was sub–divided into five phases (given in more detail below): simulation unit, execution unit, rendering engine, network infrastructure, and results gathering/experimentation. The first four phases were documented with phase reports[52, 53, 54, 55], while the final phase is documented in this dissertation.

4.6.1 NAVL Coding Phases

The coding effort was spread out over the seven months from Janurary to August of 1998. Approximately two half days were spent per week for the duration (56 half days). The NAVL prototype grew to 7398 lines of code (LoC), thus the productivity was 132 LoC/Half Day.

A rule of thumb is that the average software engineer has approximately 30 percent overhead (e.g., meetings) and produces 20 LoC/Day[41, p. 109]. This works out to

Total Coding Hours	224 Hours (Estimated)
Total Lines of Code	8912 LoC
Number of Source Files	34
Total Size of Code	1.31 MBytes
Total Object Size	7.03 MBytes
NAVL Host Working Set	1.24 MBytes

Table 4.7: Overall prototype implementation statistics

be 14.3 Loc/Half Day.

Therefore, the NAVL prototype was completed with approximately nine times the industry average productivity. This large increase in productivity can be attributed to the following:

1. There is only one developer for this project. Thus all design decisions can be made immediately, without discussion and delay–inducing meetings.

2. Much of the design decisions and implementation details were decided in advance through the efforts of writing the oral proposal (i.e., functional and design specifications).

3. Most of the design uses software techniques and tools familiar to the developer (e.g., shared memory, networking and threading).

A summary of some software metrics are provided in Table 4.7 (this includes auxiliary programs like the route mapper and demonstration programs) while each phase/task is detailed in Table 4.8.

The relatively small working set (e.g., 1.24MBytes) makes the NAVL host suitable for embedded applications such as a World Wide Web browser plug–in.

Table 4.8: Implementation statistics broken down by Phase/Task

Phase/Task Name	LoC	Duration (Weeks)	Productivity (LoC/Half Day)
Simulation Unit	1403	5	140
Execution Unit	2457	10	123
Rendering Engine	1483	10	74
Network Infrastructure	2055	4	257
Route Mapper	937	3	156

4.7 Chapter Summary

This chapter describes the implementation of the NAVL DVR system prototype. The prototype took approximately seven months to complete, with over 7300 lines of code.

The prototype implementation is limited to a LAN-only design due to the limited resources for evaluating a large scale deployment. Quite simply, there was not the manpower required to establish a full implementation with multiple LANs connected via the distributed client/server network architecture. The slimed-down implementation still allows much of the system to be evaluated such as the ForceLet concept, and overall performance. The main components are the SEU, rendering and collision detection components, and the networking subsystem and protocols.

The SEU is responsible for simulating ForceLets (for slave objects) and executing behavior routines (for master objects). The interface between the SEU and the application and object's behavior routines is made through the OPI (for objects) and the API (for applications).

These interfaces allow for the separation of the object writer and the application writer. They also allow for the encapsulation of DVR services in an implementation-variant way. It does not matter how the NAVL host services the interface's request, only that it does.

The rendering and collision detection components use the OpenGL graphics library and a nested-loop collision detection algorithm for their respective services. The OpenGL library was found to be a great asset for rapid prototyping, both within the NAVL system and independent of the NAVL system, due to the large support material available on the World Wide Web. It certainly would not have been an easy task to reproduce the wealth of rendering services available in OpenGL by scratch.

The nested-loop implementation, while not efficient, turned out to be sufficient for the needs exhibited within the NAVL prototype. Many of the applications written for NAVL have been to verify and test the effectiveness of the ForceLet coherency mechanism. In these environments, there are few objects and only minimal collision detection services required.

The networking subsystem and its protocols provide the fundamental services for transmitting events and ForceLets between NAVL hosts. A broadcast UDP mechanism was chosen for the base-level transport, with a wide array of PDUs mapped within the payload.

A route mapping technique was also provided. This allows application programs to convert between their own internal curve representations into a set of ForceLets.

Finally, the project plan for the prototype was discussed. The prototype was implemented with approximately nine times the industry average productivity. Much of this productivity is due directly to the preparation work required for the proposal.

Chapter 5

Evaluation of NAVL System Architecture

This chapter examines the components of the NAVL architecture using either a practical evaluation of the prototype, or a theoretical analysis.

The chapter begins with a bandwidth and latency analysis of several network topologies. The results of these analyses indicate that the networking architecture chosen for NAVL is the overall best choice for bandwidth and latency.

Several profile plots of the prototype running a demonstration application were created showing the movement of a target object over time. An extra delay was added for each successive plot (0ms up to 1500ms) to determine the effects of network delay on the resulting plots. To form a basis for comparison, a stream-of-data coherency algorithm was used in parallel to the ForceLet coherency mechanism and plotted.

The rendering and collision detection components are also examined. The nested-loop collision detection algorithm was seen to be a limitation, however not a dramatic one.

5.1 Network Architecture

The network architecture is one of the critical elements of the NAVL architecture. It is largely responsible for reducing total network bandwidth, reducing latency, and distributing the network load throughout the NAVL system.

Ordinarily the network architecture would be evaluated by examining the behavior of the NAVL prototype. However, due to limited resources, the NAVL prototype was limited to a LAN-only design. Many of the elements of the network architecture, such as the message managers, can not be evaluated directly due to limited resources, thus a theoretical analysis is provided here.

To give a basis of comparison, the NAVL architecture will be compared to peer-to-peer and client/server network architectures. Furthermore, network behavior is modeled with both bandwidth (network loading) and latency (network delay) models:

1. A bandwidth analysis determines the rate of messages traveling through the network as a whole, and the maximum rate of messages that travel through any one point in the network.

2. A latency analysis determines the longest expected delay in sending a message from one NAVL host to another.

5.1.1 Bandwidth Analysis

A simple model of the network deployment is created to evaluate the network effects on bandwidth and latency. The model consists of a network of hosts connected in one of the candidate topologies (e.g., peer-to-peer, client/server, or distributed client/server).

Symbol	Description
λ_i	Message spawn rate (messages/second) from each host
BW_T	The total number of messages/second handled by the network infrastructure.
BW_P	The peak number of messages/second handled by the busiest host(s) (more than one for peer-to-peer) in the network.
n	The number of hosts in the network.
ℓ	The number of message manager servers in a distributed client/server network. This is equivalent to the number of LANs in the network.
m	The number of hosts per LAN (equivalent to n/ℓ under the assumption that hosts are distributed uniformly).

Table 5.1: Symbols used in bandwidth analysis

For the bandwidth analysis, it is assumed that each host transmits messages at a constant rate λ_i ('i' for input) which traverse the network fabric with no delay. The topology is wholly responsible for the peak bandwidth and the total bandwidth (expressed as packets per second). This model is termed a logical model, as opposed to a physical model as seen in the network delay analysis section.

Table 5.1 lists the symbols used as part of the bandwidth analysis.

Peer-to-Peer (P-P)

The peer-to-peer network architecture has n hosts with direct communication links to each other. Unlike the client/server network architecture (discussed below), none of the hosts have special significance.

For each $\lambda_i n$ packets generated, they must be transmitted to all other hosts $n-1$.

Each host, however, only receives $\lambda_i(n-1)$ messages and sends $\lambda_i(n-1)$ messages.

$$\begin{aligned} \mathrm{BW}_T &= \lambda_i n(n-1) && (5.1) \\ \mathrm{BW}_P &= \lambda_i(n-1) + \lambda_i(n-1) = 2\lambda_i(n-1) && (5.2) \end{aligned}$$

Client/Server (C/S)

The client/server architecture describes the situation where a single server manages the computational duties, and many clients handle the user interaction duties. For DVR, this means that the server must manage the entire network flow for the virtual environment; all packets flow into the server for distribution to all clients.

The server receives $\lambda_i n$ messages from the clients, and then transmits $\lambda_i n(n-1)$ messages to all other clients not including the sender.

$$\mathrm{BW}_T = \mathrm{BW}_P = \lambda_i n + \lambda_i n(n-1) = \lambda_i n^2 \quad (5.3)$$

Distributed Client/Server (D C/S)

The distributed client/server network architecture is based on the regular client/server architecture, with a family of servers managing the network traffic. Special message managers replicate network packets to the low level LANs which are updated via a broadcast mechanism; only a single message manager is required per LAN. There are many similarities to the traditional client/server architecture (e.g., message manager to individual host relationship is a client/server paradigm), however there are two aspects that distinguish this from client/server:

1. There are multiple message managers acting together (e.g., a family of servers, rather than a single server); and

2. The clients of a server are defined in terms of network topology (e.g., on the same LAN) rather than logical topology.

The total bandwidth is done by counting the messages flowing through the WAN and LANs. It follows from noticing that there are $\lambda_i \ell m$ messages received by the message managers from their local LANs. The message managers send $\lambda_i \ell(\ell - 1)m = \lambda_i \ell(n - m)$ messages to the other message managers. Finally, the receiving message managers echo $\lambda_i \ell(n - m)$ messages to their local LANs.

The peak bandwidth is done by counting the messages flowing through a message manager. There are $\lambda_i m$ messages that are sent to the message manager from its local LAN. The message manager sends $\lambda_i(\ell - 1)m = \lambda_i(n - m)$ messages to other message managers. The message manager reads $\lambda_i(n - m)$ messages sent from other message managers. Finally, the message manager echoes $\lambda_i(n - m)$ messages to its local LAN.

$$
\begin{aligned}
\mathrm{BW}_T &= \lambda_i \ell m + 2\lambda_i \ell(n - m) \\
&= \lambda_i n(2\ell - 1) \qquad (5.4) \\
\mathrm{BW}_P &= \lambda_i m + 3\lambda_i(n - m) \\
&= \lambda_i n \left(3 - \frac{2}{\ell}\right) \qquad (5.5)
\end{aligned}
$$

Results Summary

The expressions derived are summarized in Table 5.2. The rank columns order the results from best (1) to worst (3).

Figure 5.1 compares the bandwidth by plotting the candidate bandwidths vs. the number of hosts. The number of LANs is assigned such that no more than 50

Name	Total BW	Rank	Peak BW	Rank
peer-to-peer	$\lambda_i n(n-1)$	2	$2\lambda_i(n-1)$	1
client/server	$\lambda_i n^2$	3	$\lambda_i n^2$	3
Distributed client/Server	$\lambda_i n(2\ell-1)$	1	$\lambda_i n\left(3-\frac{2}{\ell}\right)$	2

Table 5.2: Summary of network bandwidth expressions

hosts within a single LAN (i.e., $\ell = \lceil n/50 \rceil$). Notice how the distributed client/server network architecture is much better than both peer-to-peer and client/server for total bandwidth, and as good or better than peer-to-peer for peak bandwidth.

5.1.2 Latency Analysis

The delay estimates are performed using information regarding the topology (e.g., T_{LAN} and T_{WAN}) and queuing theory. The latency is the delay from when the message leaves the source host to when it arrives at the final destination host.

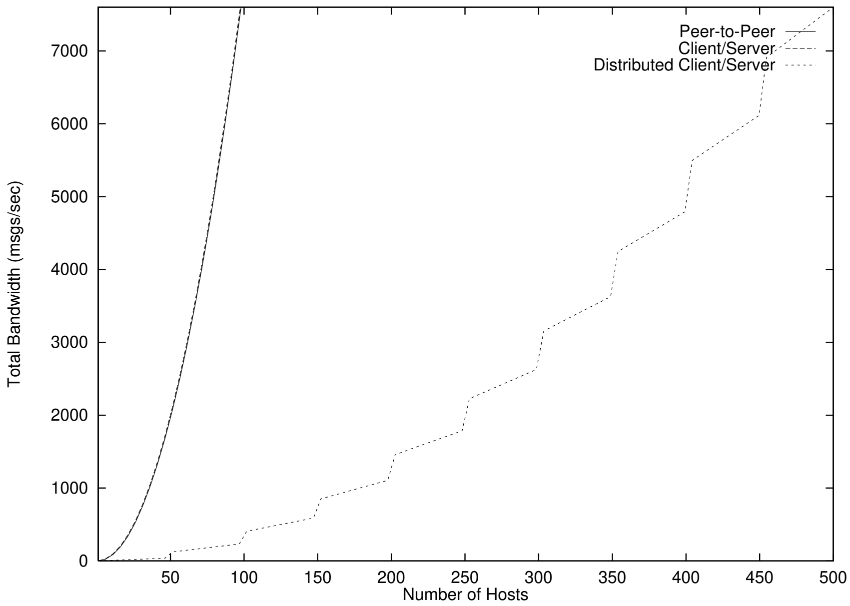
The following assumptions were made as part of the timing model:

1. Only a single message can arrive at a time due to physical network links (the router becomes a funnelling agent), and messages arrive at random times. The arrival traffic is modeled as Poisson.

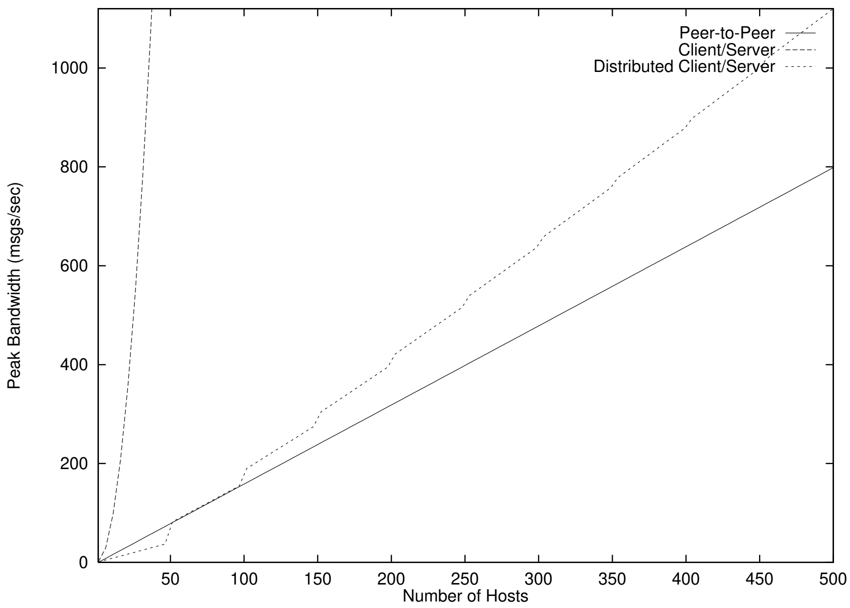
2. The servers have infinite input buffer size.

3. Processing time at the server is negligible. This analysis examines only the network architecture, not the computational structure. Thus, we will omit the processing time spent at the server handling the message.

4. There are no packet errors, thus no error detection protocols are used.



(a)



(b)

Figure 5.1: Bandwidth plots for peer-to-peer, client/server and distributed client/server network architectures: (a) the total bandwidth in the network, and (b) the peak bandwidth

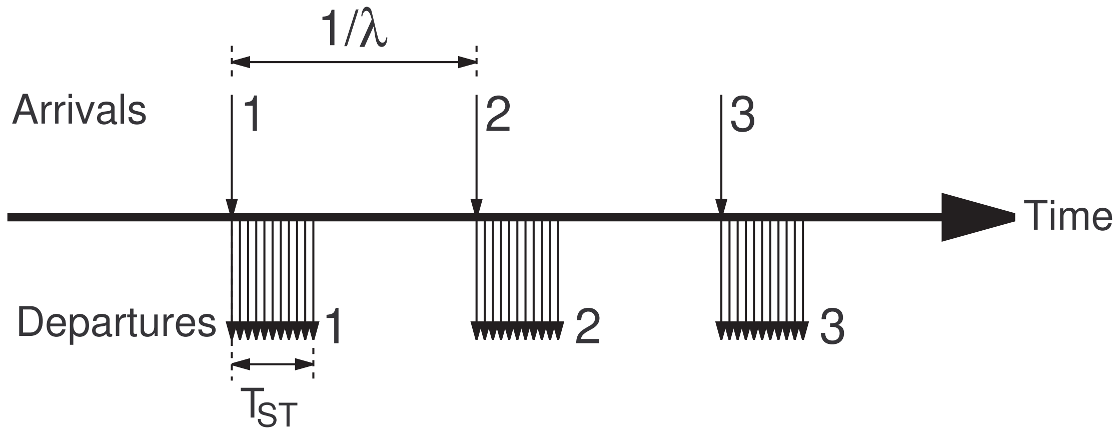


Figure 5.2: Timing diagram showing message arrivals and departures for servers. The average time between arrivals is $1/\lambda$ and the service time is T_{st}

5. When transmitting a packet to multiple hosts through the WAN, the server serializes the transmission (i.e., multicast protocols are not used). When transmitting to multiple hosts on a LAN a single packet may be sent on the LAN (i.e., broadcast protocol).

The queuing delay characteristics for the server queue in a client/server arrangement is found first and then applied to the simple client/server and the distributed client/server network architectures.

Server Queue Delay Characteristics

The server timing is modeled as shown in Fig. 5.2. The message arrivals are modeled as Poisson, and the service time is deterministic (the server spends a known quantity of time on every packet). An M/D/1 queue is used to model the traffic characteristics[22, 59, 21].

Symbol	Typical Value	Description
τ	—	Message Latency (Seconds)
λ_i	0.8 msgs/sec	Message Spawn Rate (per Host)
n	500	Number of Hosts
ℓ	10	Number of LANS (Servers)
T_{WAN}	100 ms	Transmission time on WAN
T_{LAN}	0.4 ms	Transmission time on LAN
T_{IO}	0.005 ms	I/O Wait Time

Figure 5.3: List of parameters and symbols used during latency analysis

There are n_i clients ('i' for input) sending messages with a rate of λ_i to the server producing an aggregate server arrival rate (λ) of

$$\lambda = n_i \lambda_i \tag{5.6}$$

Latency parameters are listed in Fig. 5.3, the typical value column lists the values of the parameters used during the analysis phase below.

The queuing delay for an M/D/1 queue is[22, p. 74]:

$$T_{\text{QD}} = \frac{T_{st}(2-\rho)}{2(1-\rho)} \tag{5.7}$$

where the service time is the amount of time it takes to transmit the packet to n_o hosts (where 'o' is for output), thus:

$$T_{st} = n_o T_{IO} \tag{5.8}$$

The load or utilization of the server is found as follows (where μ is the rate of messages serviced):

$$\begin{aligned} \rho &= \lambda/\mu = \lambda T_{st} \\ &= \lambda_i n_i n_o T_{IO} \end{aligned} \tag{5.9}$$

Applying Eqs. 5.7–5.9 yields:

$$T_{\mathrm{QD}}(n_i, n_o) = \frac{n_o T_{IO}(2 - \lambda_i n_i n_o T_{IO})}{2(1 - \lambda_i n_i n_o T_{IO})} \tag{5.10}$$

The delay is shown as a function of n_i and n_o to show dependence on the number of hosts sending messages to the server (n_i) and the number of hosts the server is sending to (n_o).

Network Delay Characteristics

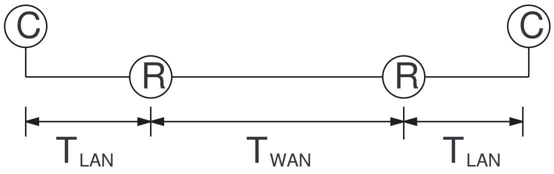
The network architectures are modeled as shown in Fig. 5.4. This may be thought of as a physical network model rather than a logical model as was used for the bandwidth analysis. The symbols in the figure are S for Server, R for Router, and C for client. The ladder-like symbol between S and R in parts (b) and (c) represents an input queue for the server.

For simplicity, we will assume the low-level network protocols (e.g., HDLC and MAC[45]) provide a consistent network platform without errors and with a constant delay between the LAN and WAN interconnects. Thus, in this model, the server input queues are the only location for delay variability.

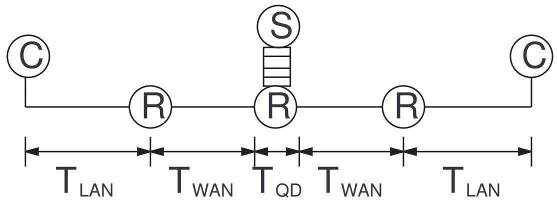
The following sections apply the queuing delay results (i.e., Eq. 5.10) to the candidate network architectures. The resulting expressions can be made by visual inspection of Fig. 5.4.

Peer-to-Peer

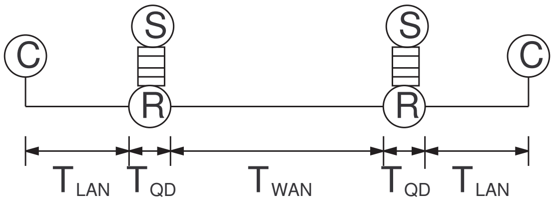
Peer-to-Peer networks do not have any server queuing delays, so the message delay is simply the transmission delay through the WAN:



(a) Peer-to-Peer



(b) Client/Server



(c) Distributed Client/Server

Figure 5.4: Physical model for latency analysis

$$\tau = 2T_{\mathrm{LAN}} + T_{\mathrm{WAN}} \tag{5.11}$$

Client/Server

A client/server system sends messages from the source host (client) to the router (a LAN interconnect). The router re-transmits the message to the router local to the server machine (a WAN interconnect). The message waits for service on the input queue, and finally gets transmitted to the client (WAN and LAN interconnects as before).

The queuing delay involves n hosts sending a message to the server, but the server only needs to transmit to $n-1$ hosts to avoid sending the message back to the originator, thus $n_i = n$ and $n_o = n - 1$. The queuing delay is:

$$\tau = 2(T_{\text{LAN}} + T_{\text{WAN}}) + T_{\text{QD}}(n, n-1) \tag{5.12}$$

Applying Eq. 5.9 with $n_i = n$ and $n_o = n - 1$ yields the load to be:

$$\rho = \lambda_i n(n-1)T_{IO} \tag{5.13}$$

Distributed Client/Server

A message in a distributed client/server architecture travels from the source host to the server (a message manager) through a LAN transmission. The server echoes the message to all other servers, which in turn echo the message to their local LANs.

The server (message manager) receives from n hosts. The message manager transmits a local message to the other $\ell - 1$ message managers and also echoes a non-local packet to the local LAN (thus $n_i = n$ and $n_o = (\ell - 1) + 1 = \ell$). The delay is found as follows:

$$\tau = 2\left(T_{\text{LAN}} + T_{\text{QD}}(n, \ell)\right) + T_{\text{WAN}} \tag{5.14}$$

Applying Eq. 5.9 with $n_i = n$ and $n_o = \ell$ yields the load to be:

$$\rho = \lambda_i n \ell T_{IO} \tag{5.15}$$

Results Summary

The expressions derived in the previous section are summarized in Table 5.3. The rank column order the results from best (1) to worst (3).

Figure 5.5(a) shows a plot of the delay vs. number of hosts when the load is set at 0.8 erlangs. The plot shows a dramatic increase in delay for the client/server architecture between 450 and 500 hosts. The distributed client/server network architecture

Name	Latency(τ)	Rank
peer-to-peer	$2T_{\text{LAN}} + T_{\text{WAN}}$	1
client/server	$2(T_{\text{LAN}} + T_{\text{WAN}}) + T_{\text{QD}}(n, n-1)$	3
distributed client/server	$2\left(T_{\text{LAN}} + T_{\text{QD}}(n, \ell)\right) + T_{\text{WAN}}$	2

Table 5.3: Summary of network latency expressions

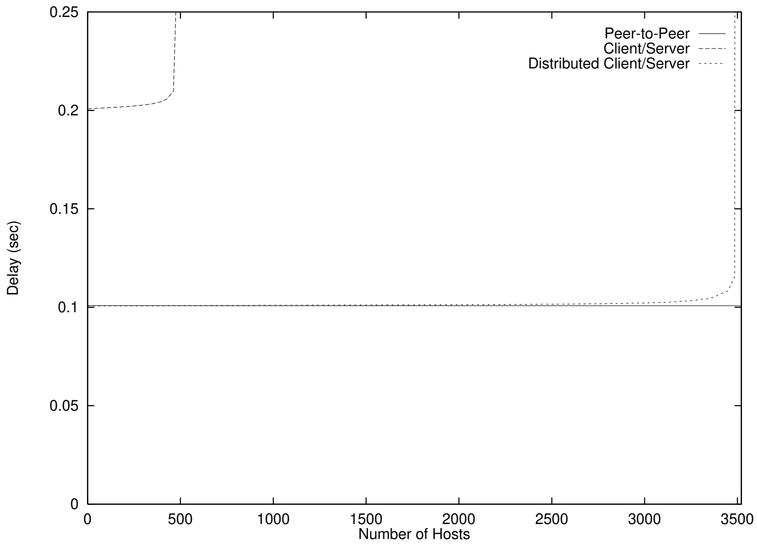
is very close to the peer-to-peer result throughout most of the plot. The message manager queing delay is only noticeable as the number of hosts approaches 3521.

Applying Eqs. 5.13,5.15 with 0.8 erlangs reveals that the maximum number of hosts supportable by the system is 500 hosts for a client/server system but is 3521 for a distributed client/server system.

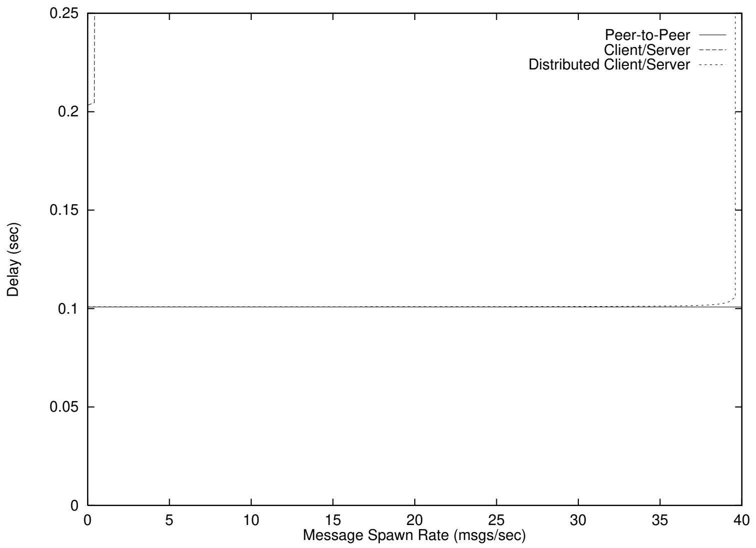
Figure 5.5(b) shows a similar result when we plot delay vs. arrival rate when the number of hosts is set to 500. In this case, the client/server network delay increases dramatically at around 0.7 to 0.8 erlangs. As with the previous plot, the distributed client/server system matches the peer-to-peer network architecture for most of the plot range. Applying Eqs. 5.13 and 5.15 with 500 hosts shows that the maximum arrival rate supported by the system is 0.8 erlangs for a client/server system but is 40 erlangs for a distributed client/server system.

5.1.3 Discussion of Results

The bandwidth analysis (i.e., Table 5.2 and Fig. 5.1) shows that the peer-to-peer network architecture is well-suited for systems with large numbers of participants (e.g., large n). The impact on any one host in the network is the smallest of any architecture. However, the amount of traffic flowing throughout the entire system is very close to the largest, which impacts the telecommunication provider who may



(a)



(b)

Figure 5.5: Delay plots for peer-to-peer, client/server and distributed client/server network architectures: (a) the delay Vs. number of hosts, and (b) the delay Vs. message spawn rate

charge extra for the communications service.

The client/server architecture fairs poorly when compared to the other architectures, making it suitable for small scale DVR applications(i.e., small n) only.

The distributed client/server architecture shows good results for both the number of messages in the network and number of messages per host. While the results indicate that the benefits increase for fewer but larger LANs, the reader is cautioned that broadcast media show very poor latencies when saturated[22]. Therefore, the deployment of a distributed client/server network architecture must ensure that network traffic within the LANs will not saturate the broadcast media.

The latency analysis (i.e., Table 5.3 and Fig. 5.5) shows that the peer-to-peer network architecture has the lowest worst-case latency due to its single hop and no queuing delay. The distributed client/server architecture outperforms the client/server architecture by a significant margin and is very close (in practise) to the peer-to-peer delay.

One major reason for the vast improvement of the distributed client/server over the client/server delay results is that the client/server requires two WAN hops to transport the message while the distributed client/server only requires one.

However, the removal of a WAN hop has been replaced by an additional queuing delay – a packet must travel through two message managers before reaching the destination. Solid state components, such as those used in buffering systems, have shown performance improvements in accordance with Moore's Law (double in performance every 18 months).

While pure communication systems have the potential to grow even faster than Moore's Law, the deployment of such technologies and the rapid increase in demand[1]

[1] The increased demand tends to absorb any improvements in network services, thereby nullifying

have prevented the realisation of a similarly impressive growth. The network topology is therefore biased toward the solid state components to leverage the growth potential of that technology.

Also, we see that the central server's bottleneck of the client/server topology causes the latency to grow much faster in the non-distributed version than with the distributed client/server. This may be observed by noting that the client/server architecture peaks at $\lambda_i = 0.8$ (when $n = 500$) while the distributed client/server architecture peaks at $\lambda_i = 40$ (also when $n = 500$).

This improvement in capacity is due to the better network distribution of the distributed client/server architecture. The message managers provide a mechanism to spread the server load over multiple places in the network. This avoids overloading any one part, as is seen with the client/server architecture.

5.2 Object Distribution and Coherency

The simulation and execution unit (SEU) is largely responsible for distributing and coordinating the actions of the master object and its slaves. The ForceLet coherency mechanism is the heart of the object distribution and coherency model. It provides the means by which the slave objects are kept in sync with the master objects.

To show the effectiveness of the ForceLet coherency mechanism as compared to stream-of-data (10Hz sampling rate), a series of prototype execution traces are plotted. Figures 5.6 to 5.11 show the Y dimension of an object being manipulated by the user in real time under the influence of varying network delays.

In the profiles, the user is moving the object up and down by applying ForceLets

any increase in network improvement.

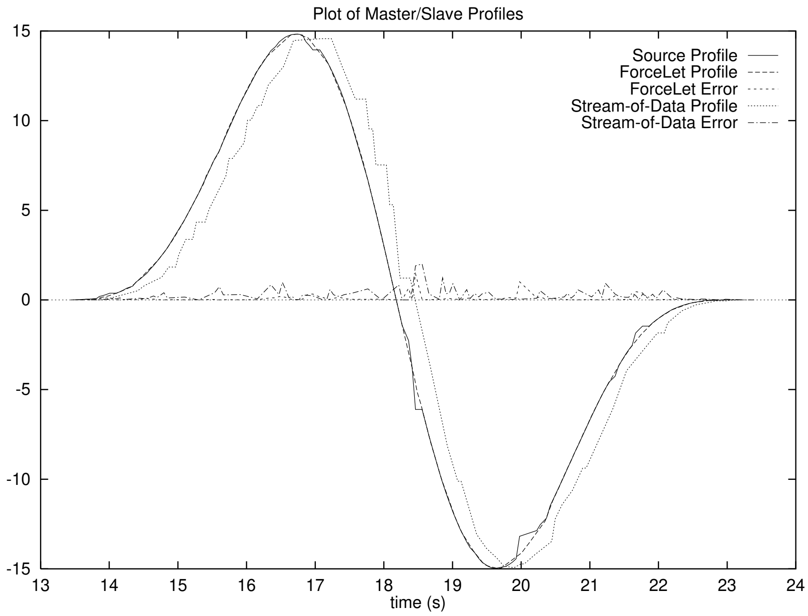


Figure 5.6: Master/Slave execution profile with no extra delay

via the keyboard. Each stroke of the keyboard sends a ForceLet to the virtual objects for a total of eight ForceLets (two up, four down, another two up).

This version of the NAVL prototype had an extra event queue to hold messages for a specified time. In this way, we are able to examine the effects of coherency under the conditions of network latency. Each movement group represents a different latency – in order they are: 0ms, 100ms, 200ms, 500ms, and 1500ms. The source and slave paths are overlayed on each other for direct comparison, and the difference between the curves are shown as an error curve.

The author's interactive entertainment experience has shown that the error rates for stream-of-data based coherency is tolerable up to approximately 100ms. From Table 5.4 we can see that the ForceLet model keeps well within the average and maximum error values (1.29 and 4.59) for all induced latency conditions tested.

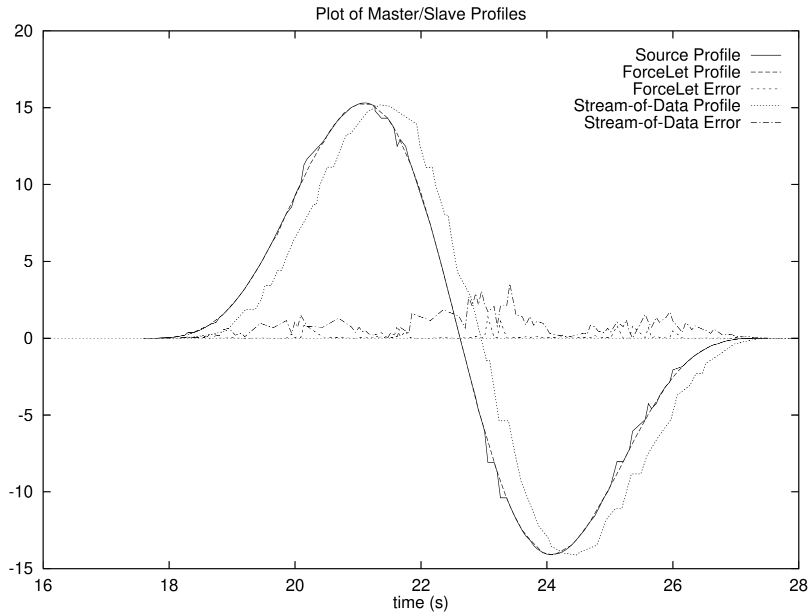


Figure 5.7: Master/Slave execution profile with 100ms delay

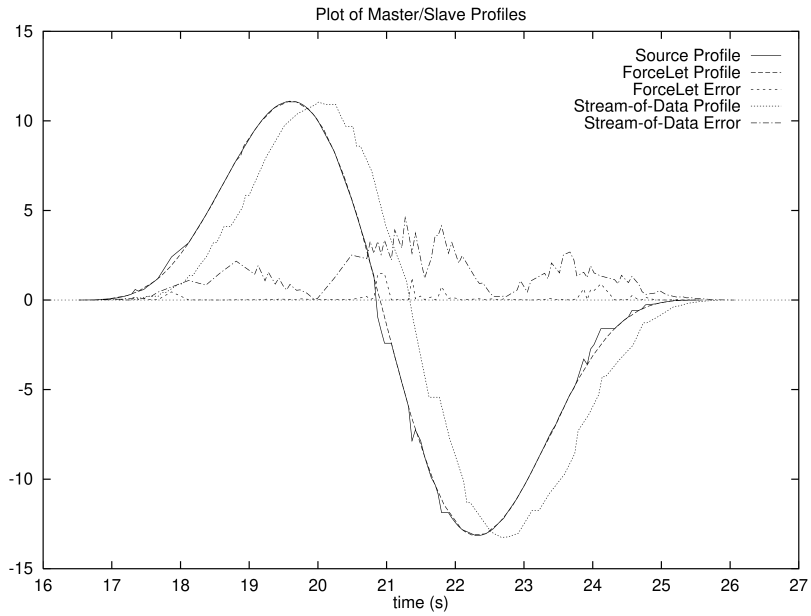


Figure 5.8: Master/Slave execution profile with 200ms delay

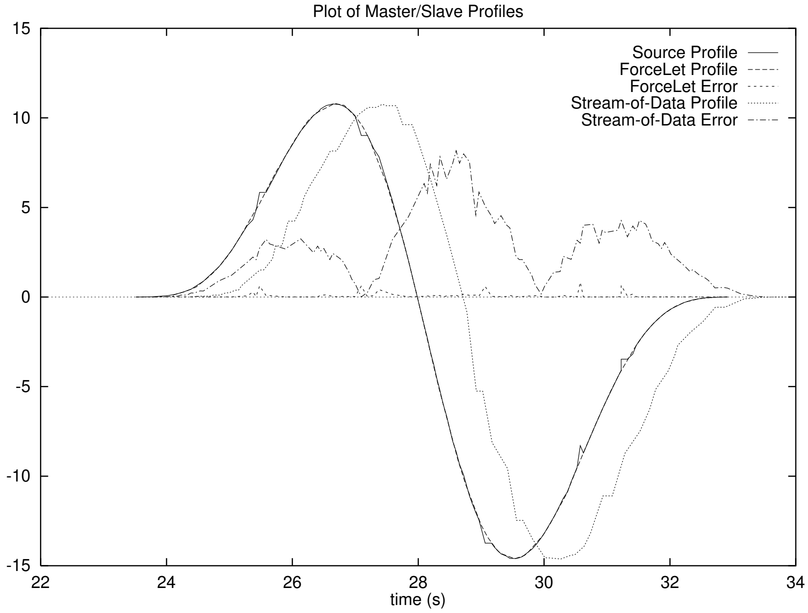


Figure 5.9: Master/Slave execution profile with 500ms delay

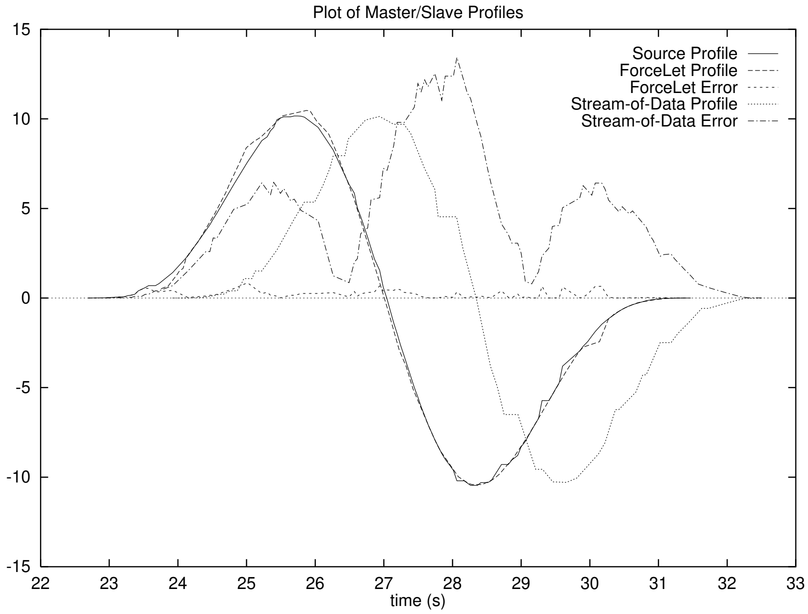


Figure 5.10: Master/Slave execution profile with 1000ms delay

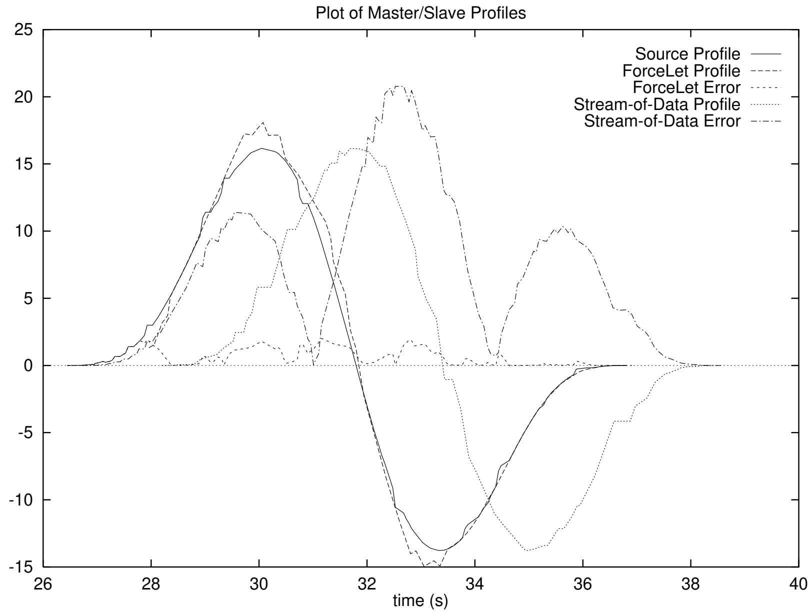


Figure 5.11: Master/Slave execution profile with 1500ms delay

Induced Latency	Stream-of-Data		ForceLet	
(ms)	Avg. Error	Max. Error	Avg. Error	Max. Error
0	0.261	2.011	0.086	1.468
100	0.692	3.545	0.107	1.398
200	1.291	4.594	0.122	1.496
500	2.805	8.167	0.069	0.609
1000	5.452	13.435	0.187	0.655
1500	9.090	20.776	0.553	2.021

Table 5.4: Errors for master/slave execution profile

Number of Virtual Objects	Average Overhead (ms)	CPU (%)	Average Framerate (FPS)
16	1	0	20
32	2	0	20
64	7	0	20
128	28	40	20
256	105	100	9.4

Table 5.5: Rendering engine and collision detection overhead. The average overhead is the amount of time spent generating the state buffer and detecting collisions per frame. The CPU percentage notes the amount of CPU required on a SPARCStation 20. Note that the frame-rate is held constant at 20 FPS until the CPU usage exceeds the systems capacity

5.3 Rendering and Collision Detection

The OpenGL graphics library provides the main rendering services in the NAVL prototype. Major increases in performance can be made by upgrading to OpenGL compatible graphics accelerators, or using improved OpenGL driver libraries. Since the majority of this work is centered on the coherency algorithm, neither of these approaches were needed. The graphics services provided by the OpenGL were deemed sufficient for our purposes.

Table 5.5 shows a table of the average overhead (in ms), CPU percentage, and frame-rate for various number of virtual objects. The virtual objects were placed such that none would touch during the experiment.

As can be seen, the NAVL prototype is capable of hosting between 128 and 256 objects. At 128 objects, there is plenty of CPU time left over for other tasks such as master object execution.

5.4 Chapter Summary

This chapter examined the three main components of the NAVL system architecture: Network Architecture, Object Distribution and Coherency Model, and the Rendering and Collision Detection Mechanism.

Network Architecture: A bandwidth and latency analysis was performed on the peer-to-peer, client/server and distributed client/server network architectures.

The distributed client/server network architecture (used by NAVL) is shown to perform at the top of the scale (or nearly top) in all cases (i.e., total bandwidth, peak bandwidth and latency).

Object Distribution and Coherency Model: A series of prototype execution trace data was plotted to determine the error between the master object and slave object under varying network delays. Both the ForceLet and stream-of-data coherency models were compared.

The results indicate that the ForceLet model is far better than stream-of-data considering both maximum error and average error. In fact, it is noted that the ForceLet model has better fidelity at 1500ms than the stream-of-data model has at 100ms.

Rendering and Collision Detection: These two components were implemented using easy to use technologies (e.g., OpenGL for rendering and nested-loop for collision detection). There was very little premium placed on tuning these components.

The results of the rendering and collision detection algorithms deomonstrate that the algorithm can achieve only approximately 128-256 objects before saturating the CPU. While these results are not sufficient for a full deployment, they served our purposes well as we typically use less than 10 objects within our applications.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

After the initial investigations into existing DVR systems, it became apparent that few researchers were looking at the entire picture. Most implementations were really just test-beds for user interfaces or VR hardware. This paradigm is often used for exploring new ideas; however, our interest is centered on designing a cost-effective DVR system.

Therefore, this research took a holistic approach to designing DVR systems. First, the state-of-the-art was examined to determine the set of problems that need to be solved in the next generation DVR systems. Then, these problems were analyzed to produce a set of six criteria for effective DVR systems (a system that meets these six criteria will fix most if not all the problems found with contemporary DVR systems). Finally, the six criteria are used as a basis for the NAVL system architecture and the design of the NAVL prototype.

The author credits this holistic approach for many successes in this project, for example:

- Many of the networking improvements came via looking at other components for opportunities to reduce network traffic. For example, the use of ForceLets in the object coherency model allows for less required network resources. Also, the use of object hierarchies allows for fewer network resources as well.
- The master/slave execution model owes much of its ease of use to the modified NAVL DVR layer diagram. By viewing DVR services in layers, it allowed us to view interactions from one system to another. For example, the design of the programming system services was made much simpler by recognizing that they could be shared by the application and master object's behaviour routine via separate interfaces. This is shown by examination of the NAVL DVR layer diagram.
- The examination of collision detection algorithms and rendering algorithms shows that there is a lot of overlap between the work carried out for these services.

6.1.1 Contributions

As part of the progression from the initial NAVL concepts to the implementation of the NAVL prototype, several contributions have been made to the state-of-the-art:

- The current DVR systems were examined. From this examination, the following were produced:
    1. DVR service layer representation,
    2. list of problems to solve, and

3. criteria for effective DVR systems.

- A holistic NAVL architecture to address the problems of existing DVR systems was created. In particular, the networking infrastructure was identified as a weak-link. In response, several strategies were developed to improve the networking layer.
- The ForceLet approach is found to be superior to stream-of-data and dead reckoning approaches.
- Autonomous objects remove the need for global network synchronization mechanisms like locking and two-phase commit.
- A prototype of the NAVL architecture was developed. The prototype was used as a platform for demonstrating and evaluating architecture features.

6.1.2 NAVL is an Effective DVR System

A DVR system that addresses the six criteria for effective DVR systems (i.e., (1) good graphics, (2) fast networking, (3) synchronized virtual events, (4) ease of use, (5) autonomous virtual objects, and (6) heterogeneous computing platforms) will also address the problems found with current DVR systems. It is felt that establishing an effective DVR system provides a step forward of the state-of-the-art of useful, practical DVR systems and helps promote more research into deploying these and better technologies in the future.

The NAVL DVR system addresses these criteria in the following ways:

Good Graphics

With the rapid ascent of 3D graphic intensive computer games, graphic acceleration hardware is now commonly available to the home user. In addition to home computers, graphics hardware is also available for many workstations such as the Sun Ultra.

The NAVL DVR prototype uses the OpenGL graphics library standard due, in part, to the large selection of graphics accelerators that follow the OpenGL standard – for both workstations and PCs.

Fast Networking

Early in the research phase, it was determined that networking played a major role for interactive virtual environments. It is very difficult to have an immersive environment if every action requires 200–500ms delay; this quickly degrades the feel of the environment.

Several elements contributed to reducing network bandwidth and reducing latency:

- A new network architecture (i.e., topology) was considered as a combination of the peer–to–peer and client/server network architectures. The distributed client/server network architecture provides many services through the message manager (e.g., filtering, name service, gateway, QoS control) as well as the overall topology. The distributed client/server topology spreads the load of the network traffic throughout the network instead of allowing it to concentrate in a single spot as seen with traditional client/server.

  Figures 5.1 and 5.5 indicate that the distributed client/server network architecture is the best overall network architecture.

- The ForceLet coherency mechanism reduces the number of updates required to keep the slave objects synchronized.

  The profiles of Figs. 5.6–5.11 were obtained with only eight ForceLets, however the stream-of-data sent well over 100 packets (approximately 10 seconds of trace data with 10Hz sampling rate).

- Object hierarchies reduce network traffic by enabling a single ForceLet to apply to an entire group of objects. Recall the articulated mannequin of Fig. 4.5. Without an object hierarchy, 10 ForceLets would be required to move the mannequin (e.g., 2 hands + 2 arms + 2 legs + 2 feet + head + torso). With the object hierarchy, a single ForceLet issued to the torso will move the entire mannequin (a savings of 10 times).

- NAVL does not require global locking strategies. Global locking strategies (e.g., two phase commit) require sending messages between remote entities to verify the requested action has occurred. The use of an autonomous object model allows NAVL to encapsulate the coordination activities to within the master object. This reduces the number of packets sent through the network, and also eliminates the characteristic blocking that happens as part of a global locking strategy.

- Autonomous objects also allow for placing the master objects close to where they have the most benefit. At present, this is done by creating the master objects on machines with high bandwidth; a future version will be able to move master objects as needed, such as to the host where the most active application resides.

Synchronized Virtual Events

The ForceLet coherency algorithm mated with the autonomous object execution model provide services for synchronized virtual events. Events can be sent between objects and the application.

Ease of Use

The DVR layering approach and, in particular, the use of interfaces (API and OPI) to segment the programming domains greatly simplify the task of designing applications for NAVL. As a contrast, a monolithic system design requires changes to many subsystem components when a new application is developed. In addition, the interactions of the ForceLets to virtual objects are performed using a Newtonian model which is easily understood by the system's user.

Autonomous Virtual Objects

As was seen earlier, autonomous virtual objects serve many purposes (e.g., reducing network traffic by reducing locking and placing master objects close to the slaves – on the same LAN if possible). Therefore, object autonomy was considered from the initial design stages as a mechanism to distribute and encapsulate units of functionality.

Heterogeneous Computing Platforms

While designed on the Solaris platform, the NAVL prototype is written using standard programming practices of the UNIX genre. It is the developer's view that it would be very simple to port the NAVL prototype to other flavors of UNIX, and would also be reaonably simple to port to 32Bit Windows platform, though the task has never been attempted.

6.2 Future Work

The NAVL system prototype can be enhanced in a number of ways. For example, there are several limitations to the prototype (e.g., LAN only, nested-loop collision detection) that should be improved.

An important marketing direction would be to provide NAVL services as a plug-in to existing browsers. Many browsers already have access to network services and 3D libraries (e.g., VRML). A valuable set of services can be added via NAVL's autonomous object distribution and ForceLet coherency algorithms.

Along these lines, the advent of Java in the past few years has transformed network computing. Java allows routines to be shipped through the networking infrastructure without concern for the target node's implementation architecture. The NAVL prototype would be improved by using Java behavior routines instead of shared libraries. This would allow the ability to load objects onto a NAVL host that does not have access to the behavior routines locally.

Extensions to the NAVL system architecture include the use of symmetric multi–processing (SMP) technologies. A symmetric multi-processing system has multiple processors connected via a common bus to a common memory system. The salient capability of SMP systems is that due to the common memory, multiple threads of execution can read and write the same data structures simultaneously. Often this reduces program complexity.

The NAVL system architecture is a prime candidate for SMP due to the existing shared memory architecture and the design of the service dispatcher. Currently, services are dispatched sequentially; however, a SMP implementation would allow multiple simultaneous thread execution. For example, the dispatcher would be able to dispatch multiple behavior routines simultaneously.

Another NAVL system architecture improvement is the ability to move master objects throughout the network. Providing this service would allow the ability to move the "heavyweight" behavior routines to NAVL hosts with powerful CPUs. In addition, it is possible to move high–use master objects closer to the locations of high–use, thereby optimizing latency for the users.

Bibliography

[1] 3dfx Interactive, Inc. The 3dfx home page. http://www.3dfx.com.

[2] Edouard André, Jean Claude Chupin, Michel Gien, Jean-Lois Grangé, Jean Le Bihan, Gérard Le Lann, Najah Naffah, Lous Pouzin, Vincent Quint, Guy Sergeant, and Hubert Zimmermann. *The Cyclades Computer Network*, volume 2. North–Holland, Amsterdam, 1982.

[3] Uyless Black. *ATM: Foundation for Broadband Networks.* Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[4] Grigore Burdea and Philippe Coiffet. *Virtual Reality Technology.* John Wiley & Sons, New York, 1994.

[5] C. Carlsson and O. Hagsand. DIVE — a multi user virtual reality system. In *IEEE VRAIS*, pages 394–400. IEEE, 1993.

[6] K. Carr and R. England. *Simulated and Virtual Realities: Elements of Perception.* Taylor & Francis, London, 1995.

[7] Glenn Case. System administrator, Informix Software, personal communication, 1998.

[8] R. Chin and S. Chanson. Distributed object–based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[9] Circle of Fire Studios, Inc. The alphaworld home page. http://www.activeworlds.com.

[10] Douglas Comer. *Internetworking with TCP/IP: Principles, Protocols and Architectures.* Prentice Hall, Englewood Cliffs, New Jersey, 2 edition, 1991.

[11] G. Coulouris and J. Dollimore. *Distributed Systems: Concepts and Design.* Addison–Wesley, Reading, Massachusetts, 1988.

[12] J. Cremer and A. Steward. The architecture of newton, a general–purpose dynamics simulator. In *IEEE International Conference on Robotics and Automation*, pages 1806–1811. IEEE, 1989.

[13] J. Crocroft and J. P. Onions. Network time protocol (ntp) over the OSI remote operations service. Technical report, University College London, June 1990. RFC-1165.

[14] IEEE standard for distributed interactive simulation — application protocols. IEEE std 1278.1–1995, IEEE, 1995.

[15] IEEE standard for distributed interactive simulation — communication services and profiles. IEEE std 1278.2–1995, IEEE, 1995.

[16] Sidney Fels and Kenji Mase. InvenTcl: A fast prototyping environment for 3D graphics and multimedia applications. In *IEEE AMCP*, pages 163–178, Japan, November 1998. IEEE.

[17] R. Fletcher. *Practical Methods of Optimization*. Wiley, Toronto, 2nd edition, 1987.

[18] Udo Flohr. Interactive online 3d's future. *IEEE Computer*, 30(7):15–16, July 1997.

[19] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison–Wesley, Reading, Massachusetts, 2nd edition, 1990.

[20] Henry Fuchs. Beyond the desktop metaphor: Toward more effective display, interaction, and telecollaboration in the office of the future via a multitude of sensors and displays. In *IEEE AMCP*, pages 30–46, Japan, November 1998. IEEE.

[21] Donald Gross and Carl M. Harris. *Fundamentals of Queueing Theory*. John Wiley & Sons, New York, 1974.

[22] Jeremiah F. Hayes. *Modeling and Analysis of Computer Communication Networks*. Plenum, New York, 1984.

[23] G. Hirzinger, J. Heindl, and K. Landzettel. Predictive and knowledge–based telerobotic control concepts. In *IEEE International Conference on Robotics and Automation*, pages 1768–1777. IEEE, 1989.

[24] Philip M. Hubbard. *Collision Detection for Interactive Graphics Applications*. PhD thesis, Department of Computer Science, Brown University, October 1993.

[25] Rick Kazman. Making WAVES: On the design of architectures for low-end distributed environments. In *IEEE VRAIS*, pages 443–449. IEEE, 1993.

[26] Jon Kuhl, Douglas Evans, Yiannis Papelis, Richard Romano, and Ginger Watson. The Iowa driving simulator: An immersive research environment. *IEEE Computer*, 28(7):35–41, July 1995.

[27] M. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, University of California at Berkeley, 1993.

[28] Tim Lindholm and Frank Yellin. *The Java Virtual Machine*. Addison–Wesley, Reading, Mass., 1996.

[29] S. Maher and J. Cohen. Virtual reality at NASA/goddard space flight center. *ACM SIGGRAPH Computer Graphics*, 30(4):49–50, November 1996.

[30] Microsoft, Inc. The directx home page. http://www.microsoft.com/directx.

[31] Sun Microsystems. Java 3d home page. http://www.sun.com/desktop/java3d.

[32] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Trans. Communications*, 10(39):1482–1493, October 1991.

[33] Brian Mirtich. Efficient algorithms for two–phase collision detection. Tech Report TR–97–23, Mitsubishi Electric Research Laboratory, December 1997.

[34] Brian Paul. The mesa 3d home page. http://www.mesa3d.org.

[35] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1990.

[36] RTIME, Inc. The rtime home page. http://www.rtimeinc.com.

[37] M. Segal and K. Akeley. The OpenGL graphics system: A specification. Technical report, Silicon Graphics, Inc., June 1992. Version 1.0.

[38] Chris Shaw and Mark Green. The MR toolkit peers package and experiment. In *IEEE VRAIS*, pages 463–469. IEEE, 1993.

[39] Chris Shaw, Mark Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with MR. *ACM Transactions on Information Systems*, 11(3):287–317, 1993.

[40] Chris Shaw, J. Liang, Mark Green, and Y. Sun. The decoupled simulation model for virtual reality. In *SIGCHI*, pages 321–328. ACM, 1992.

[41] Charles D. Sigwart, Gretchen L. Van Meer, and John C. Hansen. *Software Engineering: A Project Oriented Approach*. Franklin, Beedle & Associates, Irvine, California, 1990.

[42] Silicon Graphics. The moving worlds VRML 2.0 specification. http://www.web3d.org/VRML2.0/DRAFT2b/spec/index.html, June 1996.

[43] Bernard Sklar. *Digital Communications: Fundamentals and Applications*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[44] Trumpet Software. Trumpet winsock 4.0 home page. http://www.trumpet.com/wsock4.html, 1999.

[45] William Stallings. *Data and Computer Communications*. Prentice Hall, Upper Saddle River, New Jersey, 5th edition, 1997.

[46] Thomas Pabst. Tom's hardware guide to video boards. http://www4.tomshardware.com/guids/video.html.

[47] The virtual reality modeling language 2.0. ISO/IEC cd14772, VRML Architecture Group.

[48] VRML Consortium. The living worlds home page. http://www.vrml.org/WorkingGroups/living-worlds/index.html.

[49] Qunjie Wang. Networked virtual reality. Master's thesis, Department of Computer Science, University of Alberta, Fall 1994.

[50] J. Wann and M. Mon-Williams. Health issues with virtual reality displays: What we do know and what we don't. *ACM SIGGRAPH Computer Graphics*, 31(2):53–56, July 1997.

[51] Martine Wedlake. The NAVL distributed virtual reality system: An architecture, design, and predictive object simulation model. Ph.D. proposal, Department of Electrical and Computer Engineering, University of Victoria, December 1997.

[52] Martine Wedlake. NAVL phase 1 results: Simulation engine. Technical report, Department of Electrical and Computer Engineering, University of Victoria, February 1998.

[53] Martine Wedlake. NAVL phase 2 report: Execution engine. Technical report, Department of Electrical and Computer Engineering, University of Victoria, April 1998.

[54] Martine Wedlake. NAVL phase 3 report: Rendering engine. Technical report, Department of Electrical and Computer Engineering, University of Victoria, June 1998.

[55] Martine Wedlake. NAVL phase 4 report: Network engine. Technical report, Department of Electrical and Computer Engineering, University of Victoria, July 1998.

[56] Martine Wedlake and Kin F. Li. Sailing the high seas with the NAVL virtual reality system. In *IEEE WESCANEX*, pages 208–213, Winnipeg, M.N., May 1997. IEEE.

[57] Martine Wedlake and Kin F. Li. The simulation and execution architecture for the NAVL DVR system. In *IEEE PACRIM*, pages 93–96, Victoria, B.C., August 1997. IEEE.

[58] W. Westenhofer. Using kinematic clones to control dynamics simulation of articulated figures. Master's thesis, School of Engineering and Applied Science, George Washington University, May 1995.

[59] Michael E. Woodward. *Communication and Computer Networks: Modelling with Discrete–Time Queues*. IEEE Computer Society Press, Washington, 1994.

[60] Paul R. Woodward. Perspectives on supercomputing: Three decades of change. *IEEE Computer*, 29(10):99–111, October 1996.

Appendix A

Route Mapper C Language Implemenation

```
/*
 * Route Planner
 *
 * Martine Wedlake
 *
 * The purpose of this programme is to find the set of forcelets
 * to define a curve that hits a series of reference points.
 *
 */


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

#define SQ(x)      ((x)*(x))
#define PI         (M_PI)
#define PI_2       (2*M_PI)
#define RAND1      (((float)rand())/32767.0)
#define MIN(x, y) ((x) > (y) ? (x) : (y))
```

```
/*
 * Parameter Defines
 *
 * DELTA        The cost threshold for acceptable soln
 * MIN_GRAD     The smallest gradient for valid line search
 *              value; values less than MIN_GRAD min magnitude
 *              will produce zero lambda.
 * OVERLAP      The amount one ForceLet will overlap the
 *              next forcelet (range = 0.0 to 1.0).
 */

#define DELTA    (0.001)  /* Threshold                         */
#define MIN_GRAD (0.0001) /* Min Gradient for Linesearch       */
#define OVERLAP   0.40    /* Percent of overlap between ForceLets */

/*
 * Options Defines
 *
 * DEBUG   Shows cost and other parameters during runtime
 */

/* #define DEBUG  /* Debugging output */

typedef struct ForceLet {
    float A;    /* Amplitude  */
    float tau;  /* Start Time */
    float T;    /* Duration   */
} ForceLet_t;

typedef struct ref {
    float x;    /* Position */
    float t;    /* Time     */
} ref_t;


static void assign(ForceLet_t *out,
                   ForceLet_t *x, float a, ForceLet_t *y, int num_fl);
static float dot(ForceLet_t *x, ForceLet_t *y, int num_fl);
static void normalize(ForceLet_t *out, ForceLet_t *x, int num_fl);
```

```
/*
 * Returns the value of a sine ForceLet in the displacement
 * domain.
 */

float
val_fl(ForceLet_t fl, float t)
{
    float A   = fl.A;
    float tau = fl.tau;
    float T   = fl.T;

    if (t - tau < 0.0)
        return 0.0;
    if (t - tau > T)
        return A*SQ(T)/PI_2;

    return (float)(A*T*(t-tau)/PI_2 -
        A*SQ(T)/SQ(PI_2)*sin(PI_2*(t-tau)/T));
}



/*
 * Returns the partial derivatives of a ForceLet
 *
 *  Input : fl, num_fl, & t
 *  output: ret.A   = @fl/@A   \
 *          ret.tau = @fl/@tau  | @ is partial derivative operator
 *          ret.T   = @fl/@T   /
 *
 *  Note: While it is not necessary to return tau and T partials,
 *        for this version of the route planner, it is still
 *        usefull to have if a full improvement algorithm
 *        is to be developed that will optimise all parameters.
 */

void
deriv(ForceLet_t *ret, ForceLet_t fl, float t)
```

```
{
    float A   = fl.A;
    float tau = fl.tau;
    float T   = fl.T;

    if ((t - tau) < 0.0)
        {
        ret->A   = 0.0;
        ret->tau = 0.0;
        ret->T   = 0.0;

        return;
        }

    if ((t-tau) > T)
        {
        ret->A   = SQ(T)/PI_2;
        ret->tau = 0.0;
        ret->T   = A*T/PI;

        return;
        }

    ret->A   =  T*(t-tau)/PI_2 - SQ(T/PI_2)*sin(PI_2*(t-tau)/T);
    ret->tau = -A*T/PI_2 + A*T/PI_2*cos(PI_2*(t-tau)/T);
    ret->T   =  A*(t-tau)/PI_2 + A*(t-tau)/PI_2*cos(PI_2*(t-tau)/T) -
                A*T/(2*SQ(PI))*sin(PI_2*(t-tau)/T);

    return;
}



/*
 * Returns the value of a group of ForceLets
 *
 * ForceLets simple sum together
 */

float
```

```
fl_grp(ForceLet_t *fl, int num_fl, float t)
{
    int   i;
    float val = 0.0;

    for (i = 0; i < num_fl; i++)
    val += val_fl(fl[i], t);

    return val;
}


/*
 * Find the dot product of two ForceLet groups.  This
 * currently only considers the amplitude.
 */

static float
dot(ForceLet_t *x, ForceLet_t *y, int num_fl)
{
    int i;
    float val = 0.0;

    for (i = 0; i < num_fl; i++)
        {
        val += x[i].A*y[i].A;
        }

    return val;
}



/*
 * Utility function to provide vector assignment
 *
 * out = x + a*y
 *
 *  x & y are vectors; a is a scalar.
 */
```

```
static void
assign(ForceLet_t *out, ForceLet_t *x, float a, ForceLet_t *y,
       int num_fl)
{
    int i;
    float A;
    float tau;
    float T;

    for (i = 0; i < num_fl; i++)
        {
        if (x != NULL)
            {
            A   = x[i].A;
            tau = x[i].tau;
            T   = x[i].T;
            }
        else
            {
            A   = 0.0;
            tau = 0.0;
            T   = 0.0;
            }

        out[i].A   = A   + a*y[i].A;
        out[i].tau = tau + a*y[i].tau;
        out[i].T   = T   + a*y[i].T;
        }
}



/*
 * This function normalises a vector
 *
 * Return:             x
 *            out =  -----
 *                   ||x||
 *
```

```
 */

static void
normalize(ForceLet_t *out, ForceLet_t *x, int num_fl)
{
    float norm;
    int   i;

    norm = sqrt(dot(x, x, num_fl));

    /*
     * Now normalize it
     */

    for (i = 0; i < num_fl; i++)
        {
        out[i].A   = x[i].A/norm;
        out[i].tau = 0.0;
        out[i].T   = 0.0;
        }
    return;
}




/*
 * This computes the cost function to minimise
 *
 *  A simple sum of squares cost function
 */

float
cost_func(ForceLet_t *fl, int num_fl, ref_t *ref, int num_ref)
{
    int   i;
    float val = 0.0;

    for (i = 0; i < num_ref; i++)
        val += SQ(fl_grp(fl, num_fl, ref[i].t) - ref[i].x);
```

```
    return val;
}



/*
 * This computes the negative gradient of the cost function
 *
 * Input: fl[]  & num_fl
 *        ref[] & num_ref
 *
 * Output: ret[] & ret_mag
 *
 *   ret_mag is the dot product of the gradient with itself.
 */

void
neg_grad(ForceLet_t *ret, float *ret_mag,
     ForceLet_t *fl,  int num_fl,
     ref_t      *ref, int num_ref)
{
    int        i;
    int        j;
    ForceLet_t d;
    float      val;

    *ret_mag = 0.0;
    for (i = 0 ; i < num_fl; i++)
        {
        ret[i].A   = 0.0;
        ret[i].tau = 0.0;
        ret[i].T   = 0.0;

        for (j = 0; j < num_ref; j++)
            {
            val = ref[j].x - fl_grp(fl, num_fl, ref[j].t);
            deriv(&d, fl[i], ref[j].t);

            ret[i].A   += val*d.A;
```

```
            }
        ret[i].A   *= 2.0;        /* neg_grad --> not -2 */

        *ret_mag += SQ(ret[i].A);
        }
    return;
}



/*
 * This function projects an n-dimensional function into
 * a single dimension function along the line determined by
 *
 *   x + alpha*s
 */

static float
f_proj(float alpha, ForceLet_t *x,
                    ForceLet_t *s,   int num_fl,
                    ref_t      *ref, int num_ref)
{
    ForceLet_t *new_x;
    float       f;

    new_x = (ForceLet_t *)malloc(num_fl*sizeof(ForceLet_t));

    assign(new_x, x, alpha, s, num_fl);

    f = cost_func(new_x, num_fl, ref, num_ref);

    free(new_x);

    return f;
}




/*
```

```
 * This function finds the gradient at a point on a projection of an
 * n-dimensional function into a single dimension function along
 * the line determined by
 *
 *   x + alpha*s
 */

static float
g_proj(float alpha, ForceLet_t *x,
                    ForceLet_t *s,   int num_fl,
                    ref_t      *ref, int num_ref)
{
    ForceLet_t *new_x;
    ForceLet_t *grad;
    float       g;
    float       g_dot;

    new_x = (ForceLet_t *)malloc(num_fl*sizeof(ForceLet_t));
    grad  = (ForceLet_t *)malloc(num_fl*sizeof(ForceLet_t));

    assign(new_x, x, alpha, s, num_fl);
    neg_grad(grad, &g_dot, new_x, num_fl, ref, num_ref);
    g = -1.0*dot(grad, s, num_fl);

    free(grad);
    free(new_x);

    return g;
}




/*
 * This computes the line search
 *
 * The task is to determine the line step along the
 * search direction where x is the current position, and
 * s is the search direction.
 *
```

```
 * Hence, find lambda, s.t. f(x + lambda*s) is minimised.
 *
 * BTW: f'(x+lambda*s) = s'g(x+lambda*s) where ' is the transpose
 *                                       operator.
 *
 * This line search is based on Fletcher p. 34.
 */

float
lineSearch(ForceLet_t *x,  int num_fl,
           ref_t      *ref, int num_ref,
           ForceLet_t *s)
{
    float       f;      /* function value                   */
    float       f_p;    /* previous function value          */
    float       f_0;    /* function value @ alpha = 0       */
    float       g;      /* function derivative              */
    float       g_p;    /* previous derivative              */
    float       g_0;    /* function derivative @ 0          */
    ForceLet_t *grad;   /* gradient at x+lambda*s           */
    float       mu;
    float       alpha;
    float       alpha_p;/* previous                         */
    float       lbound; /* lower bound of acceptable point */
    float       sigma;  /* tightness parameter              */
    float       rho;    /* meaningless parameter :-)        */
    float       tau;    /* stepsize parameter               */
    float       a;      /* left bracket                     */
    float       b;      /* right bracket                    */

    lbound = DELTA;     /* typically 0.0  */
    rho    = 0.9;       /* typically 0.9  */
    sigma  = 0.05;      /* typically 0.01 */
    tau    = 1.0;       /* typically 9.0  */

    f_0 = f_proj(0.0, x, s, num_fl, ref, num_ref);
    g_0 = g_proj(0.0, x, s, num_fl, ref, num_ref);

    if (IsNANorINF(g_0))
        {
```

```
        printf("LineSearch Caught NaN or Inf!\n");
        printf("PID = %d\n", getpid());
        while (1) sleep(1);
        }

    if (fabs(g_0) < MIN_GRAD)
        {
#ifdef DEBUG
        printf("lineSearch() gradient too small; lambda = 0\n");
#endif /* DEBUG */

        a = 0.0;
        b = 0.0;

        goto found_it;
        }

    /*
     * Initialisation
     */

    mu = (lbound - f_0)/(rho*g_0);
    alpha_p = 0.0;
    alpha   = mu/16.0;        /* choose between (0, mu] */
    f_p = f_0;
    g_p = g_0;

    /*
     * Run the loop
     */

    while (1)
        {
        f = f_proj(alpha, x, s, num_fl, ref, num_ref);

        if (f <= lbound)
            {
            a = b = alpha;

            goto found_it;
```

```
        }

    if ((f > (f_0 + alpha*g_0)) ||
        (f >= f_p))
        {
        a = alpha_p;
        b = alpha;

        goto found_it;
        }

    g = g_proj(alpha, x, s, num_fl, ref, num_ref);

    if (fabs(g) <= -1.0*sigma*g_0)
        {
        a = b = alpha;

        goto found_it;
        }

    if (g >= 0.0)
        {
        a = alpha;
        b = alpha_p;

        goto found_it;
        }

    if (mu <= 2.0*alpha - alpha_p)
        alpha = mu;
    else
        {
        float low, high;
        /*
         * Choose between
         *
         *   [2*alpha - alpha_p, min(mu, alpha+tau*(alpha-alpha_p))]
         */

        low = 2.0*alpha - alpha_p;
```

```
            high = MIN(mu, alpha + tau*(alpha-alpha_p));

            alpha = low + RAND1*(high-low);

            alpha = 2.0*alpha - alpha_p;
            }

        if (IsNANorINF(alpha))
            {
            printf("LineSearch Caught NaN or Inf!\n");
            printf("PID = %d\n", getpid());
            while (1) sleep(1);
            }

        f_p = f;
        g_p = g;
        alpha_p = alpha;
        }

found_it:
    return (a + b)/2.0;
}




/*
 * Prints the current position
 */

void
print_pos(ForceLet_t *fl, int num_fl,
    float cost, float g_dot, float lambda)
{
    int         i;
    ForceLet_t *g;

    printf("---\n\n");
    printf("Cost = %f Gradient = %f Lambda = %f\n\n",
        cost, g_dot, lambda);
```

```
    for (i = 0; i < num_fl; i++)
        printf("i=%d: (A=%f, tau=%f, T=%f)\n", i,
            fl[i].A, fl[i].tau, fl[i].T);
    printf("\n\n");

    return;
}




/*
 * This function places the forcelets with a specified
 * overlap (found in the OVERLAP macro)
 */

void
init_fl(ForceLet_t *ret,
        int         num_fl,
        ref_t      *ref,
        int         num_ref)
{
    float       T;
    int         i;


    T = ref[num_ref-1].t / (float)(num_fl - (num_fl - 1.0)*OVERLAP);

    for (i = 0; i < num_fl; i++)
        {
        ret[i].A= 1.0;
        if (i > 0)
            ret[i].tau = ret[i-1].tau + (1.0-OVERLAP)*T;
        else
            ret[i].tau = 0.0;
        ret[i].T   = T;
        }
    return;
}
```

```
/*
 * This conjugate gradient method will use the
 * fletcher-reeves form, which requires only knowledge
 * of gradients.
 *
 * Input:  ref[], num_ref, & num_fl
 * Output: ret[]
 *
 * Assumptions:
 *   - The reference points are given chronologically.
 *   - The ret[] array storage is managed by the caller.
 */

float
cg(ForceLet_t *ret,  int num_fl,
   ref_t      *ref, int num_ref)
{
    ForceLet_t *g;      /* gradient          */
    ForceLet_t *d;      /* search direction  */
    float       g0_dot; /* dot product       */
    float       g1_dot; /* dot product       */
    float       beta;
    float       lambda;
    float       cost;

    /*
     * Initialise ForceLets.
     */

    g = (ForceLet_t *)malloc(num_fl*sizeof(ForceLet_t));
    d = (ForceLet_t *)malloc(num_fl*sizeof(ForceLet_t));


    neg_grad(g, &g0_dot, ret, num_fl, ref, num_ref);
    normalize(d, g, num_fl);        /* d = -g / ||-g||  */
```

```
    cost = cost_func(ret, num_fl, ref, num_ref);
    lambda = 1.0;

    printf("\n\n***** CG BEGUN *****\n\n");
    print_pos(ret, num_fl, cost, g0_dot, lambda);

    while ((cost > DELTA) && (fabs(g0_dot) > DELTA) && (fabs(lambda) > 0.0))
        {
        lambda = lineSearch(ret, num_fl, ref, num_ref, d);

        assign(ret, ret, lambda, d,  num_fl); /* fl = fl + lambda*d */

        neg_grad(g, &g1_dot, ret, num_fl,     /* get grad           */
                 ref, num_ref);
        beta = g1_dot/g0_dot;                          /* find beta          */
        g0_dot = g1_dot;

        assign(d, g, beta, d, num_fl);                  /* d = g + beta*d     */
        normalize(d, d, num_fl);              /* d = d / ||d||      */

        cost = cost_func(ret, num_fl, ref, num_ref);
#ifdef DEBUG
        print_pos(ret, num_fl, cost, g0_dot, lambda);
#endif /* DEBUG */
        }

#ifdef DEBUG
    printf("\n\n***** CG TERMINATED *****\n\n");
    print_pos(ret, num_fl, cost, g0_dot, lambda);
    printf("\n");
#endif

    if (cost > DELTA)
        fprintf(stderr,
            "ERROR: Caught in local minima: Cost = %f\n", cost);

    free(g);
    free(d);

    return cost;
```

```
}



#define GET_NEXT(x, y) if (((x) = strtok((y), " \t")) == NULL) continue;

/*
 * This reads a data file with the reference points
 */

void
readFile(char *filename,
         int *num_fl,
         ref_t **ref_x, ref_t **ref_y, ref_t **ref_z, int *num_ref)
{
    FILE       *f;
    char       *str;
    char        line[256];
    int         i;

    /* initialise */

    *num_fl  = 0;
    *num_ref = 0;
    *ref_x   = NULL;
    *ref_y   = NULL;
    *ref_z   = NULL;

    i = 0;

    if ((f = fopen(filename, "r")) == NULL)
        {
        printf("File doesn't exist\n");
        exit(1);
        }

    while (fgets(line, sizeof(line), f) != NULL)
        {
        if (((int)strlen(line) < 1) || (line[0] == ';'))
```

```
        continue;
    line[strlen(line)-1] = '\0';        /* remove trailing \n */

    GET_NEXT(str, line);        /* seed */

    /*
     * test for numfl
     *      numfl n
     *
     * Note: This must come before any ref statements
     */

    if (!strcmp(str, "numfl"))
        {
        GET_NEXT(str, NULL);
        *num_fl = atoi(str);
        }

    /*
     * test for numref
     *      numref n
     *
     * Note: This must come before any ref statements
     */

    if (!strcmp(str, "numref"))
        {
        GET_NEXT(str, NULL);
        *num_ref = atoi(str);

        *ref_x = (ref_t *)malloc(*num_ref*sizeof(ref_t));
        *ref_y = (ref_t *)malloc(*num_ref*sizeof(ref_t));
        *ref_z = (ref_t *)malloc(*num_ref*sizeof(ref_t));
        }

    /*
     * test for reference point
     *      ref <x, y, z, t>
     */
```

```
        if (!strcmp(str, "ref"))
            {
            if (*num_ref == 0)
                {
                printf("ref lines must FOLLOW numref lines\n");
                exit(1);
                }

            GET_NEXT(str, NULL);                /* X */
            sscanf(str, "%f", &(*ref_x)[i].x);

            GET_NEXT(str, NULL);                /* Y */
            sscanf(str, "%f", &(*ref_y)[i].x);

            GET_NEXT(str, NULL);                /* Z */
            sscanf(str, "%f", &(*ref_z)[i].x);

            GET_NEXT(str, NULL);                /* t */
            sscanf(str, "%f", &(*ref_x)[i].t);
            (*ref_y)[i].t = (*ref_x)[i].t;
            (*ref_z)[i].t = (*ref_x)[i].t;

            i++;
            }
        }

    fclose(f);
}



/*
 * This generates an output report
 */

void
gen_output(ref_t      *ref_x,
           ref_t      *ref_y,
           ref_t      *ref_z,
           int         num_ref,
```

```
          ForceLet_t *fl_x,    int num_fl_x,
          ForceLet_t *fl_y,    int num_fl_y,
          ForceLet_t *fl_z,    int num_fl_z,
          float       cost_x,
          float       cost_y,
          float       cost_z)
{
    int   i;
    float t;
    float t0 = ref_x[0].t;
    float T  = ref_x[num_ref-1].t;

    printf("Reference Points:\n\n");
    printf("t\tx\ty\tz\n");

    for (i = 0; i < num_ref; i++)
        {
        printf("%f\t%f\t%f\t%f\n", ref_x[i].t, ref_x[i].x,
            ref_y[i].x, ref_z[i].x);
        }

    printf("\n\nForceLets Generated by Route Planner\n");
    printf("Square Error: x=%f, y=%f, z=%f\n\n", cost_x, cost_y,
        cost_z);

    for (i = 0; i < num_fl_x; i++)
        {
        printf("fl 1 sin x %f %f %f\n",
            fl_x[i].tau, fl_x[i].T, fl_x[i].A);
        }

    for (i = 0; i < num_fl_y; i++)
        {
        printf("fl 1 sin y %f %f %f\n",
            fl_y[i].tau, fl_y[i].T, fl_y[i].A);
        }

    for (i = 0; i < num_fl_z; i++)
        {
        printf("fl 1 sin z %f %f %f\n",
```

```
            fl_z[i].tau, fl_z[i].T, fl_z[i].A);
        }

    printf("\n\nTrajectory:\n\n");
    printf("t\tx\ty\tz\n");

    for (t = t0; t < T; t += T/100.0)
        {
        printf("%f\t%f\t%f\t%f\n", t,
          fl_grp(fl_x, num_fl_x, t),
          fl_grp(fl_y, num_fl_y, t),
          fl_grp(fl_z, num_fl_z, t));
        }




}





/*
 * This function removes (prunes) ForceLets that do not
 * contribute much to the overall curve.
 *
 * Recall: the distance moved by a sinusoidal ForceLet is
 *             d = 1/(2PI)AT^2.  We can use this to
 *             put a threshold on the displacement; any
 *             ForceLet that does not move the virtual
 *             object X amount gets removed.
 *
 *  Return 1 if pruning happend, 0 if no pruning
 */

int
prune_fl(ForceLet_t *fl, int *num_fl)
{
    int i;
    int j;
```

```
    int rc = 0;

    i = 0;
    while (i < *num_fl)
        {
        if (fabs(fl[i].A) <= 0.5)
            {
            for (j = i; j < *num_fl-1; j++)
                fl[j] = fl[j+1];

            (*num_fl)--;
            rc = 1;
            }
        else
            i++;
        }
    return rc;
}



main(int argc, char *argv[])
{
    ForceLet_t *fl_x;        /* ForceLet group -- X Dimension */
    int         num_fl_x;
    ForceLet_t *fl_y;        /* ForceLet group -- Y Dimension */
    int         num_fl_y;
    ForceLet_t *fl_z;        /* ForceLet group -- Z Dimension */
    int         num_fl_z;
    int         num_fl;
    ref_t      *ref_x;       /* Reference Points -- X Dimension */
    ref_t      *ref_y;       /* Reference Points -- Y Dimension */
    ref_t      *ref_z;       /* Reference Points -- Z Dimension */
    int         num_ref;
    float       cost_x;
    float       cost_y;
    float       cost_z;

    if (argc != 2)
        {
```

```
        printf("usage: %s filename\n", argv[0]);
        exit(1);
        }

    readFile(argv[1], &num_fl, &ref_x, &ref_y, &ref_z, &num_ref);

    num_fl_x = num_fl_y = num_fl_z = num_fl;

    fl_x = (ForceLet_t *)malloc(num_fl_x*sizeof(ForceLet_t));
    fl_y = (ForceLet_t *)malloc(num_fl_y*sizeof(ForceLet_t));
    fl_z = (ForceLet_t *)malloc(num_fl_z*sizeof(ForceLet_t));

    init_fl(fl_x, num_fl_x, ref_x, num_ref);
    init_fl(fl_y, num_fl_y, ref_y, num_ref);
    init_fl(fl_z, num_fl_z, ref_z, num_ref);

    do
        {
        cost_x = cg(fl_x, num_fl_x, ref_x, num_ref);
        cost_y = cg(fl_y, num_fl_y, ref_y, num_ref);
        cost_z = cg(fl_z, num_fl_z, ref_z, num_ref);
        }
    while (prune_fl(fl_x, &num_fl_x) ||
           prune_fl(fl_y, &num_fl_y) ||
           prune_fl(fl_z, &num_fl_z));

    gen_output(ref_x, ref_y, ref_z, num_ref,
               fl_x,  num_fl_x,
               fl_y,  num_fl_y,
               fl_z,  num_fl_z,
               cost_x, cost_y, cost_z);

    free(fl_x);
    free(fl_y);
    free(fl_z);
}
```

Appendix B

NAVL Header Files

B.1 NAVL.h

```
/*
 * Simulator Data Structures
 *
 * Martine Wedlake
 *
 * NAVL system wide header file
 */

#ifndef NAVL_H
#define NAVL_H

#ifndef TRUE
#define TRUE            1
#endif /* TRUE */

#ifndef FALSE
#define FALSE           0
#endif /* FALSE */

#define NAVL_VERSION "ALPHA-1"
```

```
#define MS2SEC(x) ((float)(x)/1e3)        /* Millisecs to Secs */
#define SEC2MS(x) ((Time_t)((x)*1e3))     /* Secs to Millisecs */
#define getTimeSec() (MS2SEC(getTime()))


/*
 * NAVL structures
 */

typedef int  Bool_t;
typedef long Time_t;            /* in milliseconds */

typedef struct Id_str {
    int user:1;                 /* user or Object */
    int host:15;                /* host ID        */
    int id:16;                  /* Obj/User ID    */
} Id_t;

#define ID_HOST(x)        ((x).host)
#define ID_LOCAL(x)       ((x).id)
#define ID_EQ(x, y)       (((x).user == (y).user) && \
                          ((x).host == (y).host) && \
                          ((x).id   == (y).id))

extern Id_t null_oid;          /* found in util.c */
extern Id_t ether_oid;         /* found in util.c */

/*
 * Orientations are specified by three sets of angles: theta, gamma,
 * and lambda.  Looking from the tip of the coordinate to the origin
 * these angles rotate counter-clockwise around the Z, X, and Y
 * axes respectively.
 *
 * To specify a position in space, we must have an understanding
 * for the order of rotations that must be performed.  They are:
 * theta, gamma, and then lambda.
 *
 * Please note: The quaternion approach may be worthwhile to consider
 * for future use.
```

```
 */

typedef struct {
    float theta;                /* CCW angle along Z axis to origin */
    float gamma;                /* CCW angle along X axis to origin */
    float lambda;               /* CCW angle along Y axis to origin */
} Orient_t;

extern Orient_t base_orient;    /* The starting orientation    */
                                /* (from origin along +Z axis) */

/*
 * Coordinates are specified in the the standard right hand X, Y, Z
 * Cartesian coordinate system (X to the right, Y up, and Z toward
 * the viewer).
 */

typedef struct {
    float x;                    /* Position X */
    float y;                    /* Position Y */
    float z;                    /* Position Z */
} Coord_t;

extern Coord_t base_pos;        /* the starting position (at origin) */


typedef enum {
    X,
    Y,
    Z
} Dim_t;



typedef enum {
    RC_OK,
    RC_ERROR,
} RC_t;
```

```
typedef enum {
    MODE_INIT,
    MODE_OK,
    MODE_EXITING,
    MODE_EXITED
} Mode_t;


#define TEXT_SIZE        128


/*
 * Object Types
 */

typedef struct {
    float red;          /* between 0.0 and 1.0 */
    float green;        /* between 0.0 and 1.0 */
    float blue;         /* between 0.0 and 1.0 */
    float alpha;        /* between 0.0 and 1.0 */
} Colour_t;

typedef enum {
    SHAPE_NONE,         /* Uninitialised       */
    SHAPE_ORIGIN,       /* Origin (axis)       */
    SHAPE_SPHERE,
    SHAPE_CUBE,
    SHAPE_CONE,
    SHAPE_TORUS,        /* Doughnut            */
    SHAPE_TEAPOT,       /* The old standby     */
    SHAPE_TEXT          /* Text (string)       */
} ShapeType_t;

typedef struct {
    float radius;
} SphereDate_t;

typedef struct {
    float size;
} CubeData_t;
```

```
typedef struct {
    float base;
    float height;
} ConeData_t;

typedef struct {
    float innerRadius;
    float outerRadius;
} TorusData_t;

typedef struct {
    float size;
} TeapotData_t;

typedef struct {
    float size;           /* Size of characters */
    char  text[TEXT_SIZE]; /* String             */
} TextData_t;

typedef struct {
    ShapeType_t type;
    int         is_solid;  /* or wireform */
    Colour_t    colour;
    union {
        SphereDate_t sphere;
        CubeData_t   cube;
        ConeData_t   cone;
        TorusData_t  torus;
        TeapotData_t teapot;
        TextData_t   text;
    } data;
} Shape_t;


/*
 * Prototypes
 */

Time_t getTime(void);
```

```
#endif /* NAVL_H */
```

B.2 api.h

```
/*
 * api.h
 *
 * Martine Wedlake
 *
 * The client side API header file
 *
 */

#ifndef API_H
#define API_H

extern Coord_t  null_pos;
extern Orient_t null_orient;

typedef struct ob_state {
    Id_t     id;        /* Object Identifier             */
    Coord_t  pos;       /* Current Position              */
    Orient_t orient;    /* Current Orientation           */
    Shape_t  shape;     /* Object Shape                  */

    /*
     * pointers that relate objects to each other in a
     * hiararchy
     */

    struct ob_state *parent;
    struct ob_state *next_sibling;
    struct ob_state *prev_sibling;
    struct ob_state *first_child;

} ObjState_t;
```

```
/*
 * API Prototypes
 */

RC_t connectNAVL(int host, int *userId);
RC_t disconnectNAVL(int host);

RC_t getObjState(ObjState_t **objState,
                          int *numStates);
RC_t freeObjState(void);

RC_t newObject(Id_t *id,
               char *name,
               char *init_data,
               int   init_size,
               Id_t  ref_id);
RC_t attachObject(Id_t id);
RC_t moveObject(Id_t    id,     /* !!!DEBUG ONLY!!! */
                Coord_t pos0);  /* !!!DEBUG ONLY!!! */
RC_t freeObject(Id_t id);

RC_t sendMsg(Id_t  id,
             void *msg,
             int   msg_size);

RC_t getMode(Mode_t *mode);
RC_t getUID(Id_t *id);

/*
 * The following API calls get relayed to the Message Manager
 */

 RC_t publishObject(Id_t  oid,
                    char *name);
 RC_t retractObject(char *name);
 RC_t locateObject(char *name,
                   Id_t *oid);
```

```
/*
 * The following are used only for debugging purposes, and will not
 * be included in the "final cut" of the API.
 */

RC_t displayObject(Id_t id);
RC_t displayActiveObjects(void);

Time_t getTime(void);

#endif /* API_H */
```

B.3 opi.h

```
/*
 * Simulator Data Structures
 *
 * Martine Wedlake
 *
 * Object Programming Interface (OPI)
 *
 */

#ifndef OPI_H
#define OPI_H

#define NO_COLLISION_DETECTION 0.0

/*
 * Prototypes
 */

RC_t OPIsetTimer(Id_t   oid,
                 float  ts,
                 float  period,
                 void (*cb)(Id_t         oid,
                            EventData_t *ev_data,
                            void        *class_data));
```

```
RC_t OPIsetMsgCB(Id_t   oid,
                 void (*cb)(Id_t         oid,
                            EventData_t *ev_data,
                            void        *class_data));

RC_t OPIsetCollCB(Id_t   oid,
                  void (*cb)(Id_t         oid,
                             EventData_t *ev_data,
                             void        *class_data));

RC_t OPIrmCB(Id_t          oid,
             EventType_t   event_type,
             void        (*cb)(Id_t         oid,
                               EventData_t *ev_data,
                               void        *class_data));

RC_t OPIissueFL(Id_t  oid,
                Fl_t  fl_type,
                Dim_t d,
                float t0,
                float T,
                float a);

/*
 * if radius == NO_COLLISION_DETECTION then collision detection
 * will not be performed.
 *
 * NOTE: the OPIsetRadius() call must be AFTER the OPIsetShape()
 *       call, because the OPIsetShape() call automatically sets
 *       the radius to the boundingRadius() of the shape.
 */

RC_t OPIsetRadius(Id_t   oid,
                  float  radius);

RC_t OPIsetShape(Id_t    oid,
                 Shape_t shape);

RC_t OPIsetOrient(Id_t     oid,
```

```
                  Orient_t orient);

RC_t OPIsetPos(Id_t    oid,
               Coord_t pos);

RC_t OPInewObject(Object_t **ret_obj,
                  char      *class,
                  char      *init_data,
                  int        init_size,
                  Id_t       ref_id);

#endif /* OPI_H */
```

B.4 net.h

```
/*
 * net.h
 *
 * Martine Wedlake
 *
 * This file contains the header information for the network
 * code
 *
 */

#define PORT 1515

#define HOST_BROADCAST 0xFFFF   /* the hostid for ALL NAVL hosts */
#define HOST_MM        0        /* the hostid for MM             */

#define HOST_IS_MM(hostid)        (hostid == 1)

typedef enum {
    /*
     * NAVL Host packets
     */

    NET_REQ_OBJECT,     /* Request a slave object for attaching     */
```

```
    NET_OBJECT,         /* Slave Object (used for attaching)       */
    NET_POS,            /* Object Position                         */
    NET_ORIENT,         /* Object Orientation                      */
    NET_RADIUS,         /* Object Radius                           */
    NET_SHAPE,          /* Object Shape                            */
    NET_FORCELET,       /* ForceLet sent to Slaves                 */
    NET_MESSAGE,        /* Message event                           */
    NET_COLLISION,      /* Collision event                         */

    /*
     * Message Manager packets
     */

    NET_REQ_START,      /* MM: Request a start time new server     */
    NET_START,          /* MM: Start Time                          */
    NET_DEF_OID,        /* MM: Define (publish) V. object          */
    NET_RM_OID,         /* MM: Remove V. object                    */
    NET_REQ_OID,        /* MM: Get Published Object                */
    NET_OID,            /* MM: OID of requested object             */
} NETtype_t;


typedef struct {
    char     name[MAX_IDENT];   /* Class name                      */
    Id_t     oid;               /* Object Id                       */
    Id_t     ref_oid;           /* Reference Object Id             */
    float    mass;              /* Current mass (forcelets)        */
} NETobj_t;

typedef struct {
    Id_t     oid;               /* Object Id                       */
    Coord_t  pos;               /* Start position                  */
} NETpos_t;

typedef struct {
    Id_t     oid;               /* Object Id                       */
    Orient_t orient;            /* Current Orientation             */
} NETorient_t;

typedef struct {
```

```
    Id_t     oid;              /* Object Id                    */
    float    radius;           /* Radius (collision_detection) */
} NETradius_t;

typedef struct {
    Id_t     oid;              /* Object Id                    */
    Shape_t  shape;            /* Current Shape                */
} NETshape_t;

typedef struct {
    Id_t   oid;                /* Object ID to send ForceLet   */
    Fl_t   type;               /* Cosine or Unit Step          */
    Dim_t  d;                  /* X, Y, or Z dimension         */
    Time_t t0;                 /* Time Zero                    */
    Time_t T;                  /* Duration                     */
    float  a;                  /* Amplitude                    */
} NETfl_t;

typedef struct {
    Id_t sender;               /* OID or UID of sender         */
    Id_t recipient;            /* OID or UID of recipient      */
    char msg[MESSAGE_LENGTH];  /* Message                      */
    int  msg_size;             /* Size of message              */
} NETmsg_t;

typedef struct {
    Id_t oid1;
    Id_t oid2;
} NETcoll_t;

typedef struct {
    Id_t oid;
    char name[MAX_IDENT];      /* Search Name of object        */
} NETdef_t;



typedef union {
    NETpos_t    pos;
    NETorient_t orient;
```

```
    NETradius_t radius;
    NETshape_t  shape;
    NETobj_t    obj;                /* Object results Data          */
    NETfl_t     fl;                 /* ForceLet datagram            */
    NETmsg_t    msg;                /* Message datagram             */
    NETcoll_t   coll;               /* Collision Datagram           */
    Time_t      start;              /* MM: Start time               */
    NETdef_t         def_obj;       /* MM: Define Object            */
    char        obj_name[MAX_IDENT];/* MM: Search Name of object    */
    Id_t        oid;                /* MM: Requested OID            */
} NETdata_t;


typedef struct {
    int       to;       /* Destination host of packet   */
    int       from;     /* Host of sending packet       */
    NETtype_t type;     /* Packet type                  */
    NETdata_t data;     /* Payload                      */
} NETpacket_t;
```

B.5 clientAPI.h

```
/*
 * clientAPI.h
 *
 * Martine Wedlake
 *
 * Header file for the NAVL client drawing routines
 *
 */

#ifndef CLIENT_API_H
#define CLIENT_API_H

void changeColour(GLenum param, Colour_t colour);
void drawShape(Shape_t shape);
void drawText(void *font, float size, char *fmt_string, ...);
void drawFrame(void);
```

```
float calcFPS(void);
void initFPS(void);

#endif /* CLIENT_API_H */
```

B.6 coreAPI.h

```
/*
 * coreAPI.h
 *
 * Martine Wedlake
 *
 * Common header file for the core object class.
 *
 * This describes the data structures and shortcuts used to
 * communicate with the core object class.
 *
 * There are two modes that you can use to command the core
 * object class:
 *  - You can send a message to the core-class object (via sendMsg())
 *    with a coreAPI_t structure as the payload.  Just fill in
 *    the structure, and send the message
 *
 *  - Or you can use the shortcut functions defined in the client
 *    library to send the message for you (see the prototypes at
 *    the end).
 */

#ifndef CORE_API_H
#define CORE_API_H

typedef struct
{
    int      cmd;       /* Optional command (as defined for */
                        /* each object)                     */
    Shape_t  shape;     /* Initial Shape - type is ignored  */
    Coord_t  pos;       /* Initial position                 */
```

```
    Orient_t orient;    /* Initial Orientation              */
} coreAPI_t;

#define OB_ALL     0    /* shape, pos, and orient */
#define OB_SHAPE   1    /* only shape arg         */
#define OB_POS     2    /* only pos arg           */
#define OB_ORIENT  3    /* only orient arg        */


/*
 * Some predefined colours
 */

extern Colour_t white;
extern Colour_t black;
extern Colour_t red;
extern Colour_t green;
extern Colour_t blue;

/*
 * FE API Prototypes (definitions found in ClientLib/object.c)
 */

RC_t newOrigin(Id_t     *id,
               Id_t      ref_id,
               Coord_t   pos,
               Orient_t  orient);

RC_t newSphere(Id_t     *id,
               Id_t      ref_id,
               double    radius,
               Colour_t  colour,
               int       is_solid,
               Coord_t   pos,
               Orient_t  orient);

RC_t newCube(Id_t      *id,
             Id_t      ref_id,
             double    size,
             Colour_t  colour,
```

```
             int       is_solid,
             Coord_t   pos,
             Orient_t  orient);

RC_t newCone(Id_t     *id,
             Id_t      ref_id,
             double    base,
             double    height,
             Colour_t  colour,
             int       is_solid,
             Coord_t   pos,
             Orient_t  orient);

RC_t newTorus(Id_t     *id,
              Id_t      ref_id,
              double    innerRadius,
              double    outerRadius,
              Colour_t  colour,
              int       is_solid,
              Coord_t   pos,
              Orient_t  orient);

RC_t newTeapot(Id_t     *id,
               Id_t      ref_id,
               double    size,
               Colour_t  colour,
               int       is_solid,
               Coord_t   pos,
               Orient_t  orient);

RC_t newText(Id_t     *id,
             Id_t      ref_id,
             char     *text,
             double    size,
             Colour_t  colour,
             Coord_t   pos,
             Orient_t  orient);

#endif /* CORE_API_H */
```

VITA

Surname: Wedlake | Given Names: Martine Bruce

Place of Birth: Guelph, Ontario | Date of Birth: December 13, 1968

Educational Institutions Attended:

University of Victoria	1992 to 1999
Portland State University	1987 to 1992

Degrees Awarded:

MASc	University of Victoria	1995
BS (High Honours)	Portland State University	1992

Honours and Awards:

Dean's List	1989
Eta Kappa Nu Honour Society	1991

Publications:

M. Wedlake, K. F. Li and F. El–Guibaly. The NAVL Distributed Virtual Reality System. In *IEEE AMCP*, pages 179–195. IEEE, November 1998.

M. Wedlake and K. F.Li. The Simulation and Execution Architecture for the NAVL DVR System. In *IEEE PACRIM*, pages 93–96. IEEE, August 1997.

M. Wedlake and K. F. Li. Sailing the High Seas with the NAVL Virtual Reality System. In *IEEE WESCANEX*, pages 208–213. IEEE, May 1997.

W. R. Daasch and M. Wedlake. Rapid Layout of a Continuous–Time Transcon-

ductance–C Filter. In *International Symposium on Circuits and Systems*, pages 2256–2259. IEEE, May 1992.

W. R. Daasch, M. Wedlake, R. Schaumann, and P. Wu. Automation of the IC Layout of Continuous–Time Transconductance–C filters. *International Journal of Circuit Theory and Applications*, 20:267–282, 1992.

W. R. Daasch, M. Wedlake, and R. Schaumann. Automatic Generation of CMOS Continuous-Time Elliptic filters. *Electronic Letters*, 28:2215–2216, November 1992.