THE MULTI-CRAFT PROBLEM: A DISTRIBUTED SIMULATION APPROACH USING NETWORKED FLOATING OBJECTS

by

© Abir Zubayer

A Thesis submitted to the

School of Graduate Studies

in partial fulfillment of the requirements for the degree of

Masters of Engineering

Faculty of Engineering and Applied Science

Memorial University of Newfoundland

October, 2015

St. John's Newfoundland

ABSTRACT

The multi-craft problem is defined as simulating the interactions of multiple objects floating on water. This encompasses the direct interactions between water and the object, and indirect interactions between objects that occur via the water. Existing solutions generally treat the floating objects as simple 3-dimensional volumes with properties, such as weight and buoyancy. For many practical situations, these objects need to be simulated by complex rules. The simulation of ships is a case in point. As realistic water simulation itself is computationally expensive, accommodating the added complexity due to floating objects can be a difficult task. The research presented in this thesis proposes a method for distributed water simulation where the scope of each participating simulation is chosen by the model that governs it. For the multi-craft problem, this means simulating the water in one node and simulating the floating objects in other nodes in a network. Details of two prototypes created as part of this research are presented to show its applicability for solving this problem and how implementation of such a scheme can be achieved. Its effects on modularity, performance, scalability and reliability are also illustrated.

Key Terms: Multi-Craft Problem, Distributed Simulation, Model Development

ACKNOWLEDGEMENTS

I acknowledge my profound gratitude to my supervisors Dr. Dennis Peters and Dr. Brian Veitch for their continuous guidance and support throughout the program. Working with them has been a great learning opportunity and a wonderful experience. Their encouragement is what motivated me to finish this work successfully.

I acknowledge with thanks the financial support provided by NSERC (CREATE and Discovery Grants) and ACOA (AIF program). Thanks to the entire Virtual Environments for Knowledge Mobilization Team for their continuous assistance during the program.

Thanks a lot to Ashiq Junayed, N. M. Toufiq, Mahmudul Hasan and Debraj Laheri for being so supportive and helpful. Thanks for always being there for me and for all the fun we had together.

Special thanks to my parents A.K.M. Afzalur Rahman and Jahanara Rahman, and my wife Mashrura Musharraf for their unconditional love and support. They are my inspiration to reach new heights, and my comfort when I occasionally falter.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

BE	Big Endian
CPU	Computer Processing Unit
DDM	Data Distribution Management
DM	Declaration Management
DMSO	Defense Modeling and Simulation Office
FFT	Fast Fourier Transform
FOM	Federation Object Model
FPS	Frame Per Second
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HLA	High Level Architecture
IEEE	Institute of Electrical and Electronics Engineers
LE	Little Endian
MBD	Model Based Division
MOM	Management Object Model
MPI	Message Passing Interface
OMT	Object Model Template
PC	Personal Computer
RTI	Run Time Infrastructure
SOM	Simulation Object Model
TM	Trademark
WAN	Wireless Area Network

Chapter 1: Introduction

Solutions for water simulation, whether for scientific works [1], engineering problems [2], visual arts or computer games [3], sometimes focus on simulating water itself without accommodating the intricacies of floating object models. Solutions for interactive simulation exist [4-7] where water and floating objects affect each other, but the floating objects are mainly treated just as 3-dimensional volumes with properties, such as weight and buoyancy. For visual arts or games, complicated floating objects are sometimes animated, rather than simulated. Here simulation is defined as a process where the result is produced or observed over time based on some model, that only defines some characteristics or behavior, but the outcome is not fixed beforehand. This simplification of the floating object is generally beneficial because it frees the simulation to focus on realizing the water.

However, for many practical situations, the floating objects are more than just 3D volumes, the rules that control them are complex and simplification of their model is not an option. The multi-craft simulation is the simulation of such situations where the system needs to accommodate complex floating object models and their interactions on water. For example, in a ship simulator, the laws that govern the control of the ship, the behavior of its engine, the characteristics of its propeller and the effects of all of these on the motion of the ship, can be intricate[4, 8]. Adding this much complexity to an already complex water simulation is a difficult task and the resulting computational requirement can become enormous. In order to reduce the simulation load some realism may have to be sacrificed, like accurate interactivity.

This is certainly the case for the kinds of multi-craft scenarios that this study is interested in. Multi-ship interactions, like an ice-breaking ship leading a supply ship or a lifeboat, effects of ship wake wash on ice-fields and other ships, are instances of complex multi-craft problems that are the motivations behind this research to seek a feasible solution.

Figure 1-1 and 1-2 show two (2) examples of multi-craft scenarios. These kinds of simulations have great potential by using a virtual environment for marine personnel training, offshore emergency response training and evaluating effectiveness of operational procedures.

Figure 1-1: Multi-Craft Scenario - Ice-breaking ship leading another ship

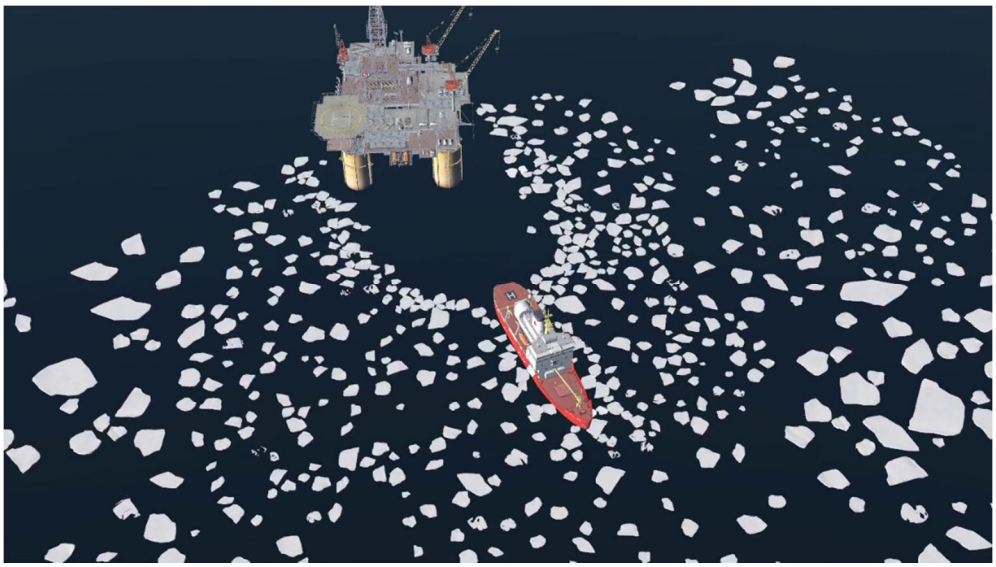


Figure 1-2: Multi-Craft Scenario - Using propeller wake wash to manage pack ice

1.1 Dividing by Model

This thesis presents a distributed simulation technique where the multi-craft simulation is divided into multiple interacting ones. The scope of each participating simulation is a design decision that can be chosen based on several factors. For example, in the multi-ship ocean simulation, if each ship has a distinct model, then the design can be to simulate each ship in an individual node and simulate the ocean in one node connected in a network. When ship and ice-field interaction is involved, we can choose to simulate the ice field as a whole on a single node and the ship and ocean on different nodes. In general, the idea is to divide the simulation into one water simulation and multiple networked floating object simulations.

A direct benefit of this technique, in contrast to a distributed simulation where each floating object also computes the surrounding water, is that we do not have to synchronize between water bodies on different nodes. This synchronization process is

non-trivial and can prove to be a significant network overhead. This technique also retains the benefits of a distributed simulation, mainly modularity. Each participating simulator, like water and floating objects, can be as complex as needed without negatively affecting others.

The goal of this thesis is to formulate this model-based distribution technique and demonstrate that such a method can be effective in solving the multi-craft problem. The main requirement is a system that provides real-time simulation of water with multiple objects having complex models. Another objective of is to observe MBD system performance and study the effect of network load on simulation throughput and design, identify, and implement system components of a general MBD system. This is done by creating two (2) prototype implementations based on this method. Chapter 3 and 4 details the implementation of these prototypes and discusses the findings. The next chapter (Chapter 2) provides background information regarding technologies used in this research such as communication architectures and water simulation algorithm, and draws an overall picture of the current advancements done in related fields. Finally, chapter 5 concludes with a discussion of the contributions of this study and some future recommendations.

Chapter 2: Background

This chapter provides background information regarding three (3) important technologies used in this research. They are: two communication platforms, a) Message Passing Interface, b) High Level Architecture, and c) an interactive water simulation algorithm, IWave. Additionally, this chapter discusses the works done in the related fields.

2.1 Message Passing Interface

The basic principle of message passing model is very simple. In the absence of global memory, it allows processes to communicate through explicit messages, like conversations between people [9]. Every message has a body and generally has tags for recipient(s) and sender's addresses, and size of the body, attached with it. By waiting for messages, processes can also be synchronized. Figure 2-1 shows a high-level description of a message passing system.

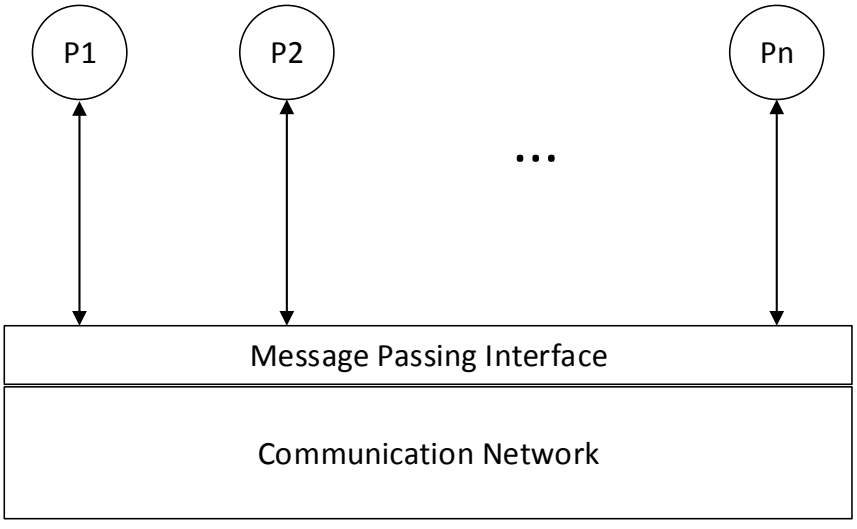


Figure 2-1: Message Passing Systems

Each node (P1 ... Pn) represents a different process and is connected to a communication network abstracted by Message Passing Interface (MPI). Nodes communicate with each other via links. Processes running on remote machines use external links and those running on the same machine use internal links.

Implementations of this model differ in choices made on multiple factors, like, whether or not messages are guaranteed to be delivered, is the delivery reliable, in order, whether multicast and broadcast are supported and should the communication be synchronous or asynchronous. For this research an implementation of the popular Message Passing Interface (MPI-2) from Microsoft Corporation is used [10].

The Message Passing Interface (MPI) Standard is a specification for a message passing library [11]. It is established to provide portability, efficiency and flexibility in writing message passing programs. Every MPI implementation is required to provide a common set of methods and features to give vendor-independence to the programmer. However, the specification doesn't force any constraint on ways to achieve them. This frees the vendors to create optimized implementations that properly utilize their hardware, and the developers won't have to worry about that, as long as they follow the standards. Its structure is simple enough to easily incorporate in any application, yet robust and complete enough to provide for the most advanced cases of message passing needs, from synchronous and asynchronous point-to-point communication to derived data types, virtual topologies, operations for global computation, synchronization and data movement. Although MPI guarantees order, it does not guarantee fairness. The programmer is responsible for avoiding starvation.

This highly popular standard has gone through multiple revisions, and the latest version is MPI-3. It has bindings for all common languages, like C/C++, Java and Fortran 2008.

2.1.1 Important MPI Concepts

MPI is a comprehensive specification providing support for a wide variety of message passing needs. The most important concepts of MPI, which are used in this research, are discussed here.

I. Communicator:

Communicator indicates a dynamic communication context for a group of processes. In MPI, data is moved from the address space of one process to the address space of another. It is important that a safe communication space is provided that guarantees unrelated messages are separate from each other. Communicators are a way to ensure that safety. There are two types of communicator. Intra-communicator is used for communication within a single group of processes. Inter-communicator is used for communication within two or more groups of processes.

MPI provides a default communicator named `MPI_COMM_WORLD`. When `MPI_Init()` is called, this defines a single context encompassing all MPI processes. New communicators can be created from this defined one using MPI provided functionalities.

II. Group and Rank:

Although communicator and group have different data types in MPI, namely, `MPI_Comm` and `MPI_Group` respectively, from a developer's perspective, there are only minor differences. They are both dynamic objects that can be created and destroyed in runtime. A group, in MPI, is an ordered set of processes. Each group is associated with a communicator. Similar to communicator, at initiation all processes belong to a single group that is associated with the default communicator `MPI_COMM_WORLD`. Groups help in organizing related processes and enable collective communication.

Processes within a group are given a unique integer identification, which is used to distinguish between different processes, called `rank`. Ranks are zero (0) based and contiguous. Processes can be part of more than one group/communicator. In each group, they have different ranks.

III. Communication Routines:

MPI provides a rich set of point-to-point communication routines. They generally all take a common set of arguments:

(1) `data_start`, `count` and `data_type`: Together they determine the location and size of the data involved in a particular communication request. The total length is calculated from the number of elements given by `count` and `data_type`. MPI supports all common data types, like `integer`, `short`, `long`, `float`, `double`, `char` and many more. The `data_start` indicates a buffer location in program space.

(2) `sender`: The rank of the sender process.

(3) `receiver`: The rank of the receiver process.

(4) `tag`: Programmer defined field to uniquely identify different messages between two processes. If unique identification is not necessary, then the wild card `MPI_ANY_TAG` can be used.

(5) `comm`: The communicator of which both the sender and receiver are parts.

The different types of communication functions supported by MPI can be divided into either 1. Blocking or 2. Non-Blocking communication. The blocking functions are of four (4) types:

a) Normal Send / Receive:

The general MPI send / receive function (`MPI_Send()` and `MPI_Recv()`) blocks until the underlying data buffer is free and can be safely overwritten. Upon return, modifying the data buffer will not affect the send / receive operation.

For `MPI_Recv()`, if the sender in not known or if it is necessary to receive from any source, then the wildcard, `MPI_ANY_SOURCE` and `MPI_ANY_TAG` can be used for sender and tag inputs respectively. In this case, the status variable will contain, among other information, the identity of the sender of that particular message.

b) Buffered Send:

In the case of normal send operation, the MPI implementation is free to adopt any strategy in ensuring buffer safety. The implementation may choose to either 1. Copy the data into temporary system buffer or 2. Wait for the corresponding receive operation to get posted. Buffered send (`MPI_Bsend()`) is a way to guarantee message buffering.

c) Synchronous Send:

This send operation (`MPI_Ssend()`) blocks until the corresponding receive operation has been posted, and the destination process has started to receive the data.

d) Ready Send:

`MPI_Rsend()`, differs from the normal send in terms of calling order, that it should only be called if the matching receive has already been posted. It is the responsibility of the programmer to ensure that. However, the blocking nature of ensuring the safety of the data buffer is the same.

Non-blocking communication happens in two steps:

Step 1: Initiation

Non-blocking or asynchronous communication allows a process to initiate a communication request that returns almost immediately and then the process moves on to execute other operations. This allows the overlap of communication and computation and has possible performance benefits. The letter 'I' is used to indicate a non-blocking send / receive operation (`MPI_Isend()` and `MPI_Irecv()`).

These two functions have an addition output parameter, `request_handle`, that uniquely identifies a previously initiated communication request.

Step 2: Completion

After an asynchronous communication has started, it is unsafe to modify the data buffer until the MPI library is done using it. It is the responsibility of the programmer to ensure that a buffer is safe to reuse. To facilitate that, MPI provides two types of routines. One is for testing the status of previously initiated communications (`MPI_Test(request_handle)`), which returns true if the operation identified by the `request_handle` is complete and `false` otherwise. The other type of routine is blocking in nature, that waits for the corresponding operation to complete (`MPI_Wait(request_handle)`).

2.1.2 Process-to-Process Synchronization

Synchronization is needed to force order of execution among parallel processes. MPI supports process-to-process synchronization in two (2) ways:

I. Blocking Calls

A process can wait for other processes by using blocking send / receive calls. The general `MPI_Send()` and `MPI_Recv()` is suitable for this purpose. The sender can make the receiver wait for a required period of time by delaying the send operation.

II. Barriers

This is a collective communication routine provided by MPI that allows for a group of tasks to be synchronized at one point. MPI barrier function looks like this,

```
MPI_Barrier(comm)
```

A process calling this function will block until all processes from the associated group of the communicator, `comm`, have called it. Then it returns, and all processes are allowed to advance.

Although not explicitly used for this research, MPI provides some other notable functionalities, such as support for virtual topology, ability to create derived data types, routines for global computation, broadcast, scatter and gather. For the most comprehensive information, the full MPI specification can be obtained from the official MPI forum [9].

2.2 High Level Architecture

High Level Architecture (HLA) is a standard for large scale distributed simulation systems that allows the creation of computer simulation out of component simulations. The participating component simulations can be geographically distributed.

One important aspect of HLA is interoperability. By conforming to HLA specification, individual simulators can become interoperable with each other and can be combined to perform large-scale computations that are beyond the scope and power of a 'single system - single program' environment. This interoperability allows the participating simulation to be written in different languages and run on different types of machines and operating systems [12].

Another beneficial aspect of HLA is reusability. Component simulators working together to achieve one particular outcome can be broken apart and re-purposed for other scenarios with little to no development overhead.

HLA is the preferred standard for modern large-scale simulation needs [13]. This open standard was first developed by the Defense Modeling and Simulation Office (DMSO) of the US Department of Defense (DoD) and later adopted as an IEEE standard [12]. It has many implementations from different vendors and programming languages, both commercial and free. For this research two freely available HLA implementations were used. One is the Pitch pRTI Free from Pitch Technologies [14], and the other is the Portico RTI from the Portico Project [15].

2.2.1 Important HLA Concepts

Figure 2-2 illustrates the different component of a general HLA system.

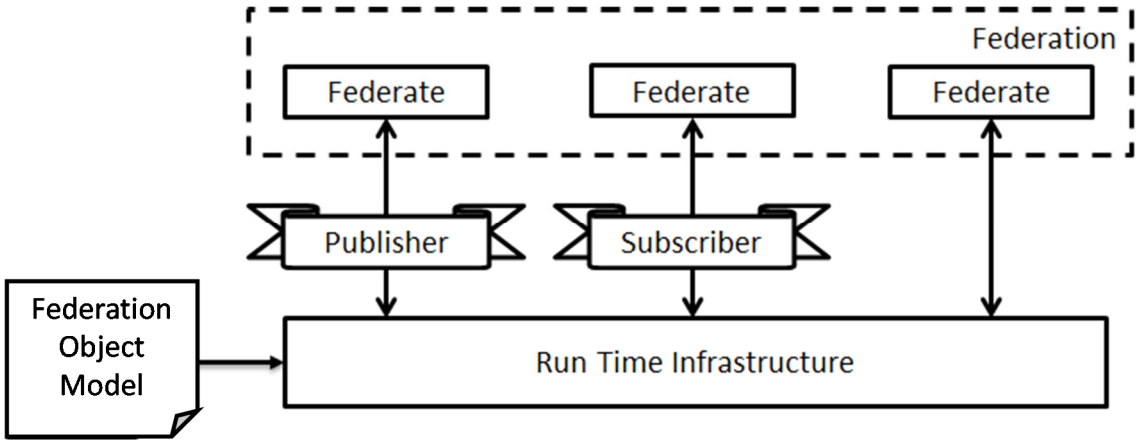


Figure 2-2: Conceptual Overview of HLA Systems

I. Run Time Infrastructure (RTI)

The RTI is a middleware that exposes the HLA specified services to simulation systems. This acts as a middle man between participating simulators. All direct communications happen between the RTI and a simulator, and it is the responsibility of the RTI to deliver the right data to the right receiver. To achieve this it provides all the necessary library and programming interfaces. HLA only defines the specification of these interfaces that frees the vendors to optimize their implementation as they see fit. For instance, the RTI itself can be distributed. This indirect communication between simulators has the possible benefit of making the system more resilient to failure, by allowing any participating simulator to crash or stop without shutting the whole simulation down.

II. Federate

Component simulators are called federates in HLA. This is any HLA compliant program that communicates with the RTI. Federate - RTI communication happens via two objects called RTI Ambassador and Federate Ambassador. Any outgoing messages from the user application, e.g., updating values, are presented to the RTI Ambassador and any callbacks from the RTI, e.g., receiving updated values, are handled by the Federate Ambassador.

III. Federation

A set of interacting federates are said to be part of a federation. These related federates share a Federation Object Model (FOM, discussed later) which is also a part of the federation.

IV. Federation Execution

Federation Execution is a single session of actual operation of a federation designed for some particular task. Each run is termed as a different federation execution.

a) Object Model Template (OMT)

OMT is a template specification identifying the format of language that can be used to describe data exchanges between federates. HLA provides a common architecture for interacting federates. However, before the interaction can happen, joined federates need to know what kind of data or services will be available on run time. OMT is the common template that federates use to specify this information and it plays a major role

in ensuring interoperability and reusability. Federation Object Model (FOM) and Simulation Object Model (SOM) are two document types that use OMT.

b) Federation Object Model (FOM)

FOM is a document describing all data requirements at runtime of a federation as a whole in a common, standardized format. Each federation has a FOM associated with it.

c) Simulation Object Model (SOM)

SOM pertains to a single federate as opposed to the federation as a whole. It describes the type and characteristic of data or services provided by individual federates to the federation.

The principal difference between FOM and SOM is that FOM focuses on inter-federate information, and SOM focuses on a federate's internal information. Among other things, these object models have three (3) main components. They are object classes, interaction classes and data types.

d) Object Classes

Object classes represent abstracted objects, whose states persist over time. For example, a ship can be described as an object class in an ocean simulation. Similar to object-oriented design, they help encapsulate related information. Object classes have attributes associated with them whose values change during a federation execution. Possible attributes for a ship object can be position, heading, and speed.

e) Interaction Classes

Interaction Classes generally represent an explicit action taken by a federate. As opposed to objects, interactions don't persist over time. Usually, they represent some discrete event, such as start, stop, and button press. Interactions may have parameters that convey more detail. The determination of what data should be classified as objects and what should be classified as interactions is not predefined and is left to the programmer.

f) Data Types

The attributes and parameters have specific data types associated with them. HLA provides many predefined data types of different bit sizes as well as complex data types like, records and arrays. In addition, HLA supports user defined data types that are particularly helpful in large-scale development.

These object and interaction class definitions also contain information about the producers and consumers of these data. A federate who offers a particular type of data is said to be the publisher and the federate who is interested in that type of data is the subscriber. The data distribution service (discussed later) provided by HLA enables the RTI to correctly propagate the data between publisher and subscriber.

2.2.2 HLA Services

HLA provides a number of services that are implemented by the RTI. A federate receives these services by making calls to the RTI through specific interfaces. These services can be categorized into eight (8) basic groups shown in table 2-1.

Table 2-1: HLA Services

Services	Main Responsibilities
Federation Management	Creation, tracking and destruction of federations and federates, synchronization point management.
Declaration Management (DM)	Keeping track of which federates are publishers and / or subscribers of each classes of data.
Object Management	Object class registration / discovery, updating object attributes and communicating interactions.
Ownership Management	Allows transfer of ownership of registered object instances between federates.
Time Management	Coordinates logical time advancement and provides consistent, orderly delivery of time-stamped data.
Data Distribution Management (DDM)	Allows advance filtering for subscribers based on data values and data regions.
Support Services	RTI start-up and shutdown, querying handles for objects / interactions.
Management Object Model (MOM)	Allows federates to query information and control the operation of the RTI.

2.2.3 Federation Development

A typical federate follows a common set of steps. Figure 2-3 shows the flow chart of a typical federate life-cycle. The `HLAModule` in both Prototype 1 and 2, developed using C++ and described in chapter 3 and 4 respectively, carry out these steps.

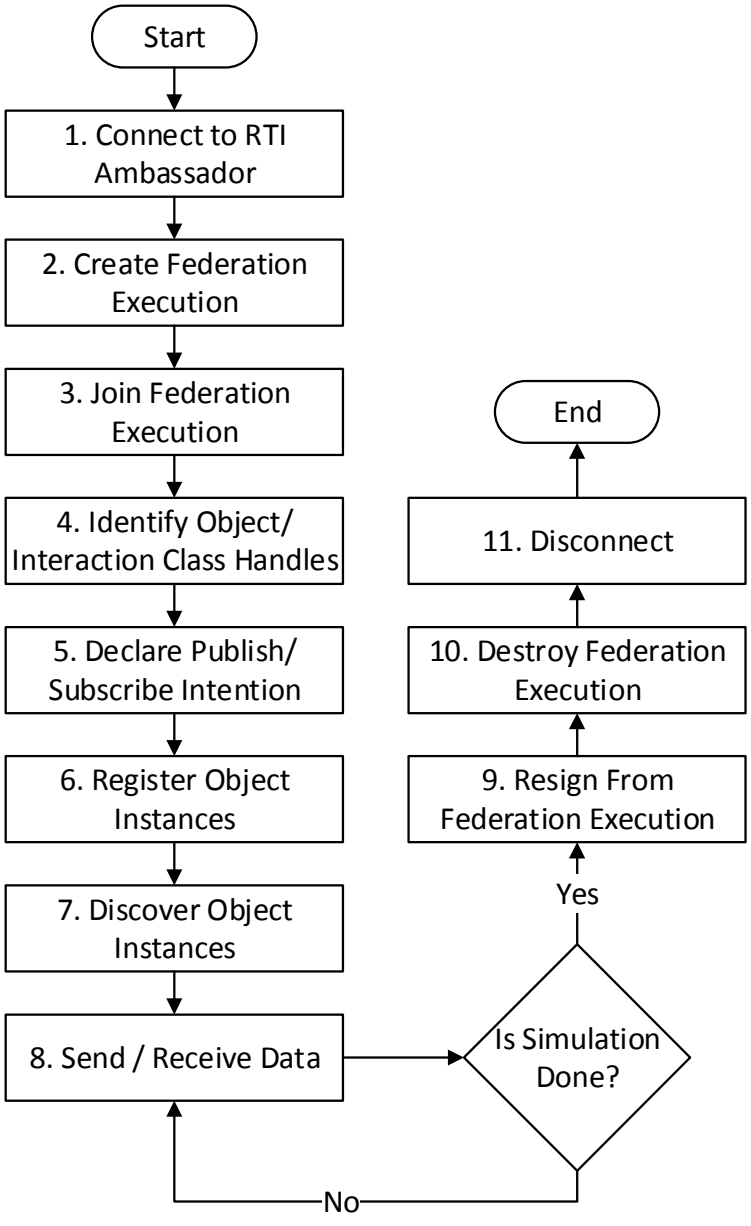


Figure 2-3: Flow Chart of Typical Federate Life-Cycle

The full HLA specification is vast and beyond the scope of this research. The specification is divided into three (3) parts. Specification regarding federation and federate rules can be found on IEEE 1516 [12]. IEEE 1516.1 [16] contains details about HLA services, and comprehensive information about OMT can be obtained from IEEE 1516.2 [17].

2.3 IWave Algorithm

IWave is an interactive water simulation algorithm developed by Tessendorf [18]. This algorithm provides a method of water simulation that is different from the more traditional Fast Fourier Transform (FFT) approach [19], where visually realistic water interactivity is difficult to obtain without loss of frame-rate. IWave's high performance is attributed to its ability to achieve interactivity through only 2D calculation on a grid, where a general fluid simulation requires 3D processing. This algorithm supports objects of any shape that can create disturbances on the water surface, such as ripple, wake, and wave reflection. The second prototype developed for this research, discussed in chapter 4, uses this algorithm for water simulation. The implementation of this method can be found within the source code of that prototype in the Supplementary Files. This section summarizes the formation and key elements of the algorithm that is used in the implementation and presents the pseudo code.

2.3.1 Linearized Bernoulli's Equation

The IWave procedure is based on the linearized Bernoulli's equation on a 2-dimensional grid of this form,

$$\frac{\partial^2 h(x,y,t)}{\partial t^2} + \alpha \frac{\partial h(x,y,t)}{\partial t} = -g\sqrt{-\nabla^2} h(x,y,t) \quad (1)$$

Here,

$h(x,y,t)$ is the height of the water surface at position (x,y) on the grid at time t,

$\frac{\partial^2 h(x,y,t)}{\partial t^2}$ represents the vertical acceleration of the wave,

$\alpha \frac{\partial h(x,y,t)}{\partial t}$ is a velocity damping term where α is a constant,

g is the acceleration due to gravity and

$\sqrt{-\nabla^2}$, defined below, is an operation that conserves the total water mass, i.e., when the height of one point of the surface rises, height of nearby regions drops.

The time derivative in equation (1) can be written as a finite difference. But before that we need to calculate the vertical derivative $\sqrt{-\nabla^2}$.

$\sqrt{-\nabla^2}$ can be written as a linear operator,

$$\sqrt{-\nabla^2} \equiv \sqrt{-\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2}} \quad (2)$$

Implemented as a convolution on a height grid it becomes,

$$\sqrt{-\nabla^2} h(x,y) = \sum_{k=-P}^{P} \sum_{l=-P}^{P} G(k,l) h(x+k, y+l) \quad (3)$$

Here,

$G(k,l)$ is an element of the convolution kernel. The kernel is a 2-dimensional square with size $(2P+1) \times (2P+1)$. According to Tessendorf (2004), $P = 6$ is a good value to provide realistic wave propagation with acceptable computation time. With $P = 6$ the size of the kernel becomes $169\ (= 13 \times 13)$ elements. The equation for $G(k,l)$ is,

$$G(k,l) = \sum_n q_n^2 e^{-q_n^2} J_0(q_n \sqrt{k^2 + l^2}) \Big/ G_0 \, , where\ 1 \le n \le 10000 \quad (4)$$

Where,

$G_0 = \sum_n q_n^2 e^{-q_n^2}$, is used to scale the center value to one ($G(0,0) = 1$),

$q_n = n \times 0.001$,

$J_0(x)$ is the Bessel function, which is available in C++ standard math library.

This kernel has two important properties,

a) The kernel values do not change throughout the simulation, so they can be calculated and stored in a 2-dimensional array at initialization.

b) The kernel values are laid symmetrically. Specifically:

$$G(k,l) = G(l,k)\ and$$

$$G(k,l) = G(-k,-l) = G(k,-l) = G(-k,l)$$

This leads to an optimization of equation (3) to reduce the number of multiplications needed.

$$\begin{aligned}\sqrt{-\nabla^2}h(x,y) = h(x,y) \\ + \sum_{k=0}^{P}\sum_{l=k+l}^{P} G(k,l)(h(x+k,y+l) \\ + h(x-k,y-l) + h(x+k,y-l) \\ + h(x-k,y+l))\end{aligned} \tag{5}$$

Now that we have all the elements, we can convert equation (1) into a finite difference resulting in,

$$\begin{aligned}&h(x,y,t+\Delta t) \\ &= \frac{(h(x,y,t)\times(2-\alpha\Delta t) - h(x,y,t-\Delta t) - \sqrt{-\nabla^2}h(x,y,t)\times g\Delta t^2)}{1+\alpha\Delta t}\end{aligned} \tag{6}$$

2.3.2 Energy Source

IWave algorithm supports wave generating sources by simply an addition operation at the beginning of each simulation loop.

$$h(x,y,t) += s(x,y,t) \tag{7}$$

Here, The source grid, $s(x,y,t)$ is the amount of relative force at position (x,y) at time t. This can be negative, positive or zero. Zero denotes no additional disturbance, and negative / positive values can be used to push or pull the wave surface at that location. $s(x,y)$ should be reduced over time if a constant force is not desired.

2.3.3 Obstruction

In IWave, obstructions are defined in a 2-dimensional grid, $o(x, y)$ of the same size as the water surface grid. In this grid, a 0 represents presence of an obstruction and 1 denotes absence of any. At the boundary of the obstructions the values $0 < o(x, y) < 1$ are used to create an anti-aliasing effect. To get interaction between water and obstruction (e.g. reflection) we just have to use a multiplication operation:

$$h(x, y) *= o(x, y) \tag{8}$$

2.3.4 Wake

A movement of a floating object is simulated by updating the obstruction grid as the object moves, by putting 0 (or $> 0\ and\ < 1$ in case of boundary) where it arrives and putting 1 at locations where it no longer exists. To enable wake, we also need to update the source grid in accordance with the current position of the object using this equation,

$$s(x, y) = 1 - o(x, y) \tag{9}$$

If the obstruction has an anti-aliased boundary, updating the source this way produces wake when the obstruction moves. In this method, the subsurface shape of an object does not affect the produced wake and if two obstructions have the same area on the 'obstruction grid' they produce the same wake. This simplification of achieving water-object interactivity by 2-D computation alone contributed to faster prototyping and analysis of the proposed system.

2.3.5 Pseudo Code

```
01 // Initialization
02 -----------------
03 Populate Convolution kernel (equation (4))
04
05
06 // Simulation loop --- START
07 ---------------------------
08
09 Update source and obstruction according to the scenario status and
   objects' positions

10 // Incorporate source and obstruction into height
11 h(x,y)+=s(x,y)
12 h(x,y)*=o(x,y)
13
14 Update vertical derivative (equation (5))
15
16 Calculate new height (equation (6))
17
18 // Simulation loop --- END
19 ---------------------------
```

2.4 Related Works

The previous three (3) sections in this chapter offer background information about existing technologies and research that the current study directly utilizes. This section discusses related works that do not directly contribute to the present research, but collectively provide an overall picture of the recent advancements made in relevant fields and informs us about best practices. One such area is fluid simulation, which has been an active research area for many years [20, 21]. In recent times, the advances in Graphics Processing Units and their applicability in parallel computing have encouraged several researchers to use GPUs for such simulations [22]. In addition to optimization in fluid simulation, the multi-craft problem can be related to multiple domains of research, most notably, the simulation of interaction between water and rigid bodies, and simulation of

floating objects with physically based, complex models. Below is a discussion of some recent advancements in each of these areas.

2.4.1 GPU Optimization

Modern Graphics Processing Units are efficient at computing a large amount of data in a parallel fashion, mainly in the context of visual rendering and image processing. It is the advent of the programmable shaders and floating point support that enabled GPUs to be used in general purpose computing and not just in the graphical rendering domain. Unlike CPUs, which have a limited number of cores and a limited number of threads that they can simultaneously support, GPU core and thread counts typically range in the thousands [23]. This gives GPUs a clear advantage for executing a common set of instructions on different elements of a large data set in parallel. Offloading data parallel compute intensive works to GPU can result in magnificent speedups [24] that, otherwise, would not be possible by CPU optimization alone. Because of this, many researchers have worked on optimizing fluid simulation by harnessing the power of GPU. Although the Navier-Stokes equations are known for accurately predicting motion of viscous fluids, the lack of a closed-form solution [25] has led to various techniques of approximating them. Among the multitude of fluid simulation methods, some of the popular techniques that have been subjected to GPU optimization are: a. Smoothed Particle Hydrodynamics (SPH), b. Lattice Boltzmann Method (LBM), and c. Eulerian grid-based method.

Smoothed Particle Hydrodynamics (SPH) [26] is a Lagrangian method that is used to simulate fluid like motion in computer graphics by discretizing the liquid volume into a set of particles that represent mass, and have properties such as position and

velocity. A “smoothing kernel function” evaluates these properties for each particle using information from neighboring particles. This method is inherently parallel, comparatively less data dependent, allows for large time steps, and provides simpler mechanism for conservation of mass, which makes it easier to be real time than the computation of Navier-Stokes equations [26]. Taking these benefits into consideration, Wu et al. [27] have implemented the SPH method completely on GPU programmable shader.

In [27], they proposed a new method that starts by grouping the grid node based on normal direction of the obstacle surface. From there, two fragment shaders are used to calculate the pressure and velocities around obstacle surfaces for static and dynamic obstacles, as well as for surfaces with and without drag. The experimental results presented show that this method allows arbitrary user defined boundary conditions. By utilizing the fragment shader instead of vertex shader, which provides higher parallelism due to having more pipelines, they improved the computational performance within GPU.

To reduce the number of rendering passes, they combined the particle properties, such as velocity, density, and temperature, directly into a single RGBA-4 channel, which allows the fragment shader to compute all of them together in one pass. Moreover, they have found that Jacobi Iteration for solving systems of linear equations is comparatively better suited for parallelization. Experimental data showed that their GPU optimization, using GeForce FX5950 Ultra with 8 fragment pipeline, achieved a significant speedup of about 14 (fourteen) times over CPU implementation.

Instead of using just one GPU, Zhang et al. [28] have shown that a multi-GPU configuration can produce even better performance. Like the study done by Wu et al. [27], the method used by Zhang et al. is also based on SPH. However, they use a

"modified Tait equation" instead of traditional ideal gas equation to avoid "high compressibility and implausible simulation results," and the neighborhood search in SPH is optimized by an index sort that reduces GPU memory overhead and searching time by utilizing parallelization provided by the Nvidia CUDA architecture.

The use of multiple GPUs, four (4) in this case, presents new problems in terms of load balancing, because data transfer between GPUs is a major bottleneck. To alleviate this, they have used a dynamic and distributed load balancing scheme, where the simulation domain is split into slices based on particle count and computation time, and each participating GPU is responsible for some consecutive slices. Time cost of particle data exchange is reduced by storing them according to slice structure, and in each step, based on the knowledge of the previous time steps, particles are selected for exchange between GPUs to reduce data transfer and ensure efficient load distribution in the future. Furthermore, cost of data exchange between GPUs are reduced by parallelizing calculation and data exchange. The experimental results presented in section 6 of their paper showed that the multi-GPU configuration can achieve a speedup of up to three (3) times compared to a single GPU system.

The **Lattice Boltzmann Method** [29] is another approach for fluid simulation that is particularly useful for handling boundary conditions and solid-fluid interfaces, because it can model both the microscopic (individual particles) and mesoscopic (probabilistic interaction of a group of particles) behavior. Although, it suffers from poor scalability and restrictive time steps, it benefits from the simplicity of the parallelization of its algorithm, making it suitable for GPU optimization. A recent study done by Rinaldi et al. [30] successfully implemented the LBM method on GPU using Nvidia's CUDA architecture.

Their work greatly benefits from the Coalesced Memory Access technique supported by the CUDA programming model (version 3.2), where a group of 16 threads (called, half-warp) can get global memory data together, in a single access, and store them in the shared memory. This technique reduces execution time by reducing global memory accesses, which is generally about 100 times slower than shared memory. After copying the global memory data, their algorithm does all calculations in the shared memory and finally writes the results back to the global memory. Coalesced Memory Access, together with their proposed "reversed advection-collision scheme," allowed them to implement the LBM method as a single step, instead of the traditional two-step algorithm [31].

One important issue that minimizes the advantages of coalesced access is the effect of code branching from conditional statements. Branches can make threads of the same group to diverge, forcing serialization and increasing the total number of operations, which especially affects the boundary cell calculation. To minimize branching effects, instead of having a lot of codes on each branch, they use "shifting indexes" that are dependent upon the boundary condition, to allow the same code to be run on each cell. Performance comparison of their implementation of the algorithm, using NVIDIA GTX 260 with 192 steam processors, simulating fluid flow in a lid driven cavity shows about 130 times speedup compared to CPU based implementation.

The Eulerian method of fluid simulation is discussed in the next section about Interactive Water Simulation.

2.4.2 Interactive Water Simulation

Interactive water simulation allows floating objects to interact with the water surface, creating ripples and wakes from the objects' movement, and position / orientation changes on the objects from forces such as waves, currents, buoyancy, and gravity. This is a complex problem, and achieving realistic interaction can be challenging. The intricacy of the geometric shape of the floating objects adds to that. One way to handle complex geometric objects and to generalize them is to think of them as being made up of much smaller and simpler shapes put together. A recent method for interactive water simulation based on SPH, developed by Ricardo da Silva Junior et al. [32], uses a "modified version of the depth peeling" [33] algorithm to discretize both the simulated fluid volume and rigid bodies into sphere shaped particles. This discretization is done at the initialization state. The radii of the particles are chosen according to the resolution requirement of a particular simulation run. All fluid particles get a fixed radii, and all rigid body particles get a separate, but also fixed, radii, which greatly simplifies collision testing at runtime. For fast searching and computation of particles, their method uses different hash tables for fluid, dynamic rigid and static rigid particles, and uses a mapping function to correlate their positions from one hash table to another.

The main contribution of [32] is a collision detection method that utilizes a heterogeneous architecture of Multi-core CPU and GPU. In their method, the collision detection step is divided into two phases: a. Broad Phase and b. Narrow Phase. "Broad Phase" is where a Multi-core CPU is used to do computationally simplistic collision testing using only 3-D bounding boxes, instead of particles. This phase identifies the

subset of particles that have a non-zero collision probability, and all other particles are filtered out, and do not play any role in the next phase of the calculation. In "Narrow Phase," an accurate, computationally expensive, and parallelized collision detection is done on GPU using CUDA architecture. Only the particles selected in the Broad Phase are considered, which increases performance significantly, because typically the percentage of particles that have a possibility of colliding is small compared to the total particle pool. After the Narrow Phase, particles that are positively identified as colliding go through "collision resolution and resultant force / torque integration." The study showed that the optimized collision detection using heterogeneous architecture is much faster (about 7 times) than GPU-bound systems alone, and can provide visually realistic rigid-fluid interaction.

An important caveat for SPH based methods discussed above, is that creating smooth liquid surfaces can be difficult. In the **Eulerian Grid-Based Method,** achieving smooth surface representation is relatively easy. In this method, instead of treating the fluid as a large number of particles, fluid properties, such as densities and velocities, are calculated as a field for the whole simulation region [34]. The tradeoff is poor scalability. This is because as the simulated region grows, the computation time grows exponentially. A technique proposed by Cohen et al. [35] tries to alleviate this shortcoming by dividing the simulation domain into two regions for each rigid body. The "near-field region" is centered on the object of interest, and fluid inside this region is governed by the velocity field calculated by Eulerian method, providing high fidelity. If the object moves, this region moves with it. Beyond this lies the "far-field region," where fluid motion is governed by particles following simple Newtonian dynamics with gravity, momentum,

drag. Their study showed that, in practical implementation, the boundary between the near-field and far-field regions is not visually discernable. By effectively restricting the computationally expensive Eulerian method inside a certain region instead of using an enormous simulation grid for the whole rendered area, and using GPU optimization, they were able to achieve realistic rigid-fluid interactivity with superior visualization, within real-time performance.

A similar approach of using high-fidelity fluid simulation only near the surface of the water to capture detailed fluid motion, and a crude estimation of water volume far away, is also adopted by Chentanez et al. [36]. However, their implementation uses “tall cells to generalize a height field underneath the water surface” that approximates the overall fluid volume. On top of it lies 3-Dimensional grids of cubic cells, that are governed by Eulerian grid based method to produce accurate velocity and pressure fields.

Rigid-fluid interaction is achieved by incorporating “solid fractions” [37] in the pressure equation, which denote the percentage of a cell covered by any solid. Effects of object movement are transferred to the water surface of these cells by the “blending of solid and fluid velocities” based on the corresponding solid fraction. Conversely, objects' positions and orientations are updated from combining all forces and torques resulting from buoyancy, drag, and gravity. Buoyancy and drag calculations depend on the solid fraction of the cells, and relative density and relative velocity of the solid and fluid. The study presents multiple large-scale scenarios having two-way rigid-fluid coupling achieving a speedup of up to 14 times compared to previous studies.

The work in this thesis uses an interactive water simulation technique developed by Tessendorf [18]. The main benefit of this technique is achieving water-object

interactivity by 2-Dimentional computation alone, unlike the works presented above that primarily focus on 3-D calculation. This greatly simplified development and allowed faster prototype development. A detailed discussion of this technique is presented in section 2.3 IWave Algorithm.

2.4.3 Modeling Intricate Floating Object

Intricate floating objects are not just rigid bodies. They are controlled by non-trivial laws and exhibit complex behaviors. Studies involving them give us insights into the extent of complexity a floating object can possess. These can be the simulation of ships, lifeboats, and ice-fields, for example. Any kind of marine vehicle simulation itself is a broad topic, containing rules that govern its controls, models of various types of engine behaviors, effects of propeller and rudder on its motion and so on.

Ueng et al. [8] have discussed a computation model for ship motion that allows the user to adjust the characteristics of the ship, such as, its size, engine power, and rudder, and environmental elements, such as, frequencies, amplitudes and directions of wave, current and winds, and observe its behavior. The model supports 6 (six) degrees of motion of the ship, which are computed separately using Newtonian dynamics, and then superimposed to generate the overall ship motion. Heave, pitch and roll are calculated using the wave heights sampled from a grid around the ship, where heave depends on the average height field, and pitch and roll are based on the differences of height fields between the top and bottom half, and left and right half of the grid, respectively. Surge and yaw are modeled using the engine power and rudder position, and sway is the result of current and wind. Drag forces and gravity are also included in the calculations. The

simulation output presented showed that the model produces visually realistic results for different kinds of ship and weather conditions.

A more advanced study to model the ship-ice interaction is done by Lubbad et al. [38]. Their numerical solution classifies ice floes that come into contact with the ship's hull, based on “comparison between the lateral area of the floe and its thickness squared.” Floes having larger lateral area are considered breakable and selected for further processing. All other floes, including those that did not collide with the ship's hull are flagged as unbreakable. Breakable ice floes can produce new, smaller floes depending on the amount of stress they are subjected to. These stresses are calculated using the “theory of Semi-Infinite Wedge-Shaped Beams on Elastic Foundation,” and if above critical level, produces cracking pattern and ultimately new floes. Lubbad et al. also introduced GPU optimization to the process by using PhysX, which is a physics engine middle-ware that can accelerate the calculation of contact detection, resolution and force calculation for thousands of solids using the CUDA architecture on supported GPUs.

A study that utilizes distributed simulation using High Level Architecture is done by McTaggart et al. [39] to simulate ship Replenishment At Sea (RAS). Using multiple participating simulators (federates) to compute the physics based modeling of replenishment gear (such as evaluation of cable tension and payload location) as well as motion and helm of the supply and receiving ship, they are able to determine whether undesirable events such as replenishment gear malfunction and payload immersion in water would occur during operation. Their simulation includes accurate mathematical model for wave induced ship motion and RAS equipment. Although hydrodynamic interaction between two ships in close proximity during Replenishment At Sea (RAS)

operation is not considered. Another research based on High Level Architecture is the Virtual Ships VS STANAG [40], which is a simulation architecture developed by the NATO Naval Armaments Group NG6. In addition to HLA specification [40] also provides specifications for virtual ship rules, development process, repository, organization and management. The primary goal of this architecture is to allow multi-national re-use and interoperability of simulation of ship and maritime acquisition.

More recently Bastin et al. [41] have shown that ice management can be done using propeller wake wash, and also used PhysX to solve the Newtonian dynamics of the ice floes. The numerical model presented produces a velocity field initiating at the propeller, which depends on the propeller diameter, speed, and thrust coefficient. GPU implementation using PhysX enabled the simulation to handle scenarios involving hundreds of ice floes.

Chapter 3: Prototype 1: Towing Simulation

3.1 Objectives

The main objective of this prototype is the study and understanding of different communication architecture that could be applicable in a multi-craft scenario. For this, a simplified version of the multi-craft problem is selected for development, where a car is dragging a dead weight using a rope or chain.

Figure 3-1: Towing Simulation

In this scenario, the rope is analogous to water, and the car and load are analogous to floating objects. Two versions of this prototype were implemented for two different communication architectures. Version 1 uses the Message Passing Interface (MPI), and version 2 uses the High Level Architecture (HLA). The information gathered for this prototype regarding MPI and HLA is consolidated in section 2.1 Message Passing Interface (MPI) and in section 2.2 High Level Architecture (HLA), respectively. This chapter provides the system analysis and discusses the output observations.

The full source code of this prototype is included in the Supplementary Files.

3.2 System Analysis

To test the selected communication architecture the system is divided into three separate but interacting processes. These three processes simulate a car, a rope and a weight respectively. Each process has its own class (car, rope and load) with its own physics logic, update and rendering cycles. The underlying communication layer enables them to pass event information to each other. The car and load only directly communicate with the rope process and not with each other, and events get acted upon whenever they are seen (received through inter-process communication), not when they are actually generated in the producing process. Each process runs on a pre-fixed 30 cycles per second frame-rate and in each cycle they first attempt to communicate with each other if necessary, then update their internal status, and lastly perform a rendering of themselves.

To keep an emphasis on the experimentation of MPI and HLA, the physics that govern each of these processes are chosen to be simple, but not too much so that it can generate enough events to test the behaviors of the communication layers. For the same reason, the GUI was also kept minimal: instead of 3D, 2D rendering was used and each process has its own separate rendering window. This is because combining rendering information from multiple processes to show on a single window is non-trivial and would have taken a significant development time without adding much to the main objective of this research.

3.2.1 Car and Load Simulator

The physics of the car and load are similar in that they both have mass and produce acceleration or deceleration depending upon whether an external force is acting

upon them or just drag, using the equation $F = ma$, where F is force, m is mass, and a is acceleration. This acceleration (or deceleration) is used in calculating the velocity ($v = u + at$) and displacement ($s = ut + \frac{1}{2} at^2$) in each iteration, where t is time, v is velocity after time t, s is the distance traveled in time t, and u is the start velocity. Displacement is used to determine the rotation of the wheels to give the feeling of a moving vehicle. So the rotation of the wheels is proportional to the forces acting on the vehicles. The only difference is that the car has an engine that can produce force, where the load has none. In both cases, whenever no force is present, drag makes them stop eventually.

3.2.2 Rope Simulator

The rope is a long semi-rigid body (not spring) with a defined length. Whenever its current length is less than this maximum, it acts as if nothing is attached with its endpoints and from the car's point of view, it is free to move on its own forces. However, when it reaches its maximum length, it acts to transfer force and mass. At that point, the car sees the presence of the load and must divide its engine's force between them.

3.2.3 Information Flow

Since all three components are simulated in three different processes, all events must be transferred through inter-process communication. Figure 3-2 shows the conceptual communication diagram of this system.

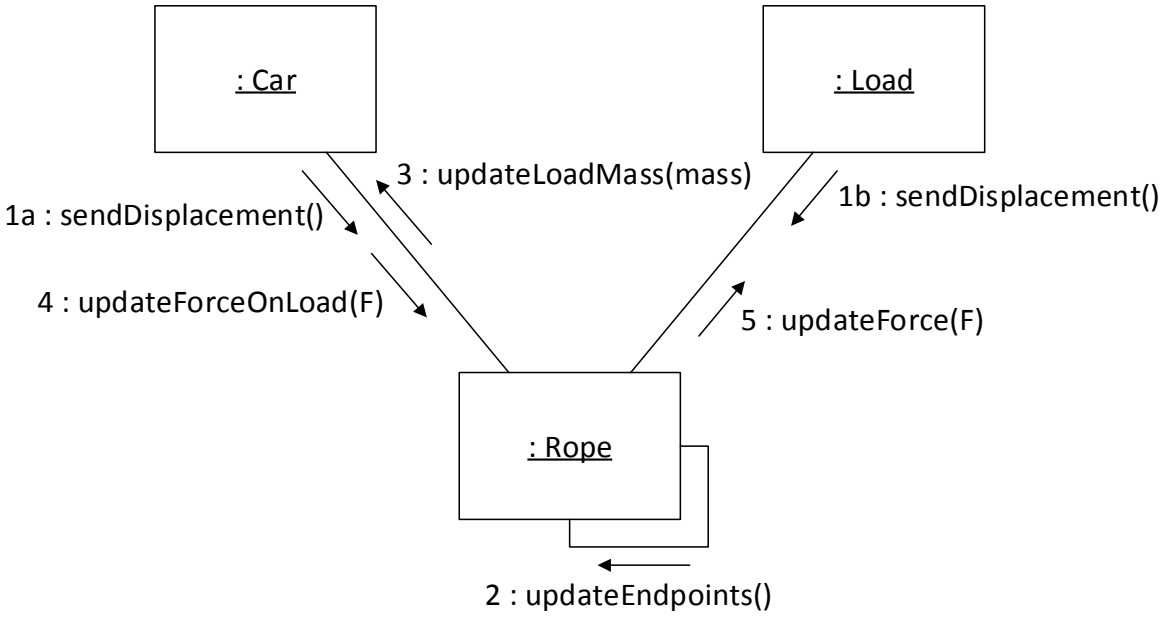


Figure 3-2: Prototype 1 Communication Diagram

There are a total of four (4) types of communication requirements.

I. Displacement events

On each iteration, whenever the car and the load are moving, and they transfer their displacement values to the rope. Both car and load avoid sending zero displacement events. Upon receiving the data the rope updates its respective endpoints position and checks to see whether or not maximum length is reached.

II. Mass Values

Whenever the rope reaches its maximum length, it transfers the mass of the load to the car, which it acquires at the beginning of the program (it assumes that the mass of the load doesn't change over time). Up until now, the car saw an additional mass of zero units, but now it receives a non-zero value and calculates the force that should act upon

that mass (and sends it to the rope), as a result the net force acting upon itself reduces and the car experiences deceleration.

III. Forces

Upon receiving force information from the car the rope transfers that to the load, the load then updates its velocity and displacement. And all this time, displacement events from both the car and load keep the rope's endpoints updated.

IV. Synchronous vs. Asynchronous

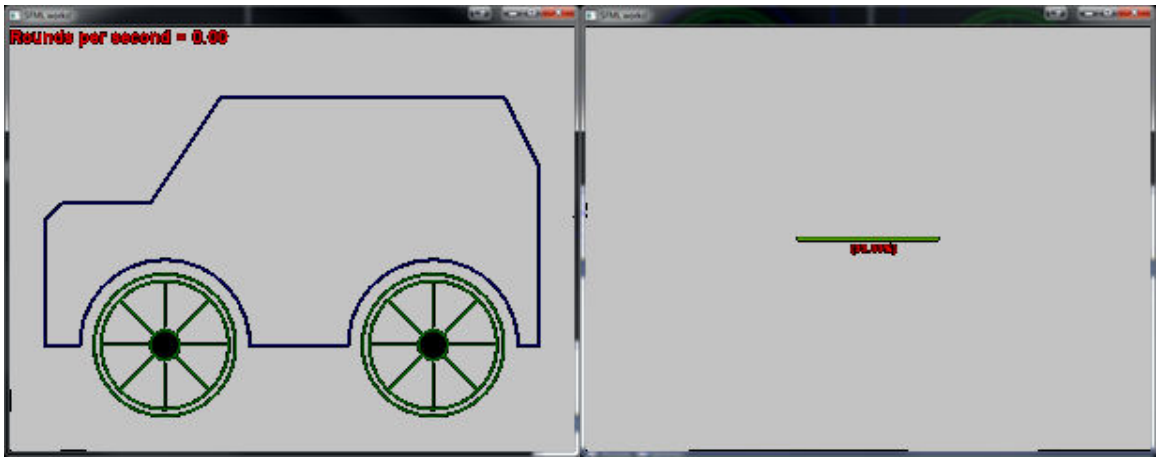
As it can be seen, the rope has to communicate with both of the other processes. To avoid blocking and unresponsiveness, Asynchronous Message Passing is used throughout the execution of the programs. This allows the rope to converse simultaneously with both the car and the load.

3.3 Implementation

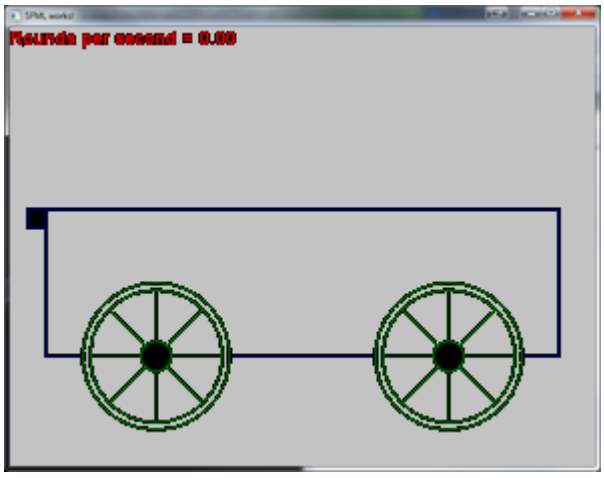
3.3.1 Version 1 : Message Passing Interface

For Microsoft Windows®, the stable implementation of MPI is 1.0.3 that implements MPI-2 [10]. The development utilizes the asynchronous send / receive functions `MPI_Isend()` and `MPI_Irecv()`. For testing the status of these operations the corresponding `MPI_Test()` function is used. All three processes are single threaded. Data transferring, simulating and checking for transfer successes are done in one sequential loop. Additionally, all communications are direct, from one specific

process to another, unlike what we will see in the HLA implementation where communication is indirect and happens via the Run Time Infrastructure. The figure below shows the graphical output of the MPI version of the prototype. Here, each window represents a separate process.



(a) Car **(b) Rope**



(c) Load

Figure 3-3: Prototype 1 (MPI) Graphical Output

3.3.2 Version 2 : High Level Architecture

I. FOM

The FOM for the HLA implementation defines two (2) object classes and two (2) interaction classes. Table 3-1 shows the publish / subscribe matrix of object classes and table 3-2 lists the interaction classes.

Table 3-1: Prototype 1 - Object Classes - Publish / Subscribe Matrix

Object Class	Attributes	Data Type	Update Type	Publisher	Subscriber
Car	PositionX	HLAfloat64BE	On Change	Car	RoapAnd Load
Car	TransferredForce	HLAfloat64BE	On Change	Car	RoapAnd Load
RopeAnd Load	LoadMass	HLAfloat64BE	Static	RoapAnd Load	Car

Table 3-2: Prototype 1 - Interaction Classes - Publish / Subscribe Matrix

Interaction Class	Order	Publisher	Subscriber
AtMaxLength	Receive	RoapAndLoad	Car
BelowMaxLength	Receive	RoapAndLoad	Car

The HLA version had to combine the rope and load processes into one "`RopeAndLoad`" process due to the limitation of the RTI (discussed later) used at the time. Because of the 1D nature of the output the car and load could only move in one axis, thus the `PositionX` is a single 64 bit float. All data is communicated on change by the RTI except the `LoadMass`, which does not change throughout federation execution and simply needs to be communicated once, at initialization.

The two interaction classes denote the two states of the rope: a) when it is fully stretched and b) when it is loose. As described previously, the Car process needs this information to determine when to exert force on the load. Since they are events and do not contain persistent information, they are represented by interactions.

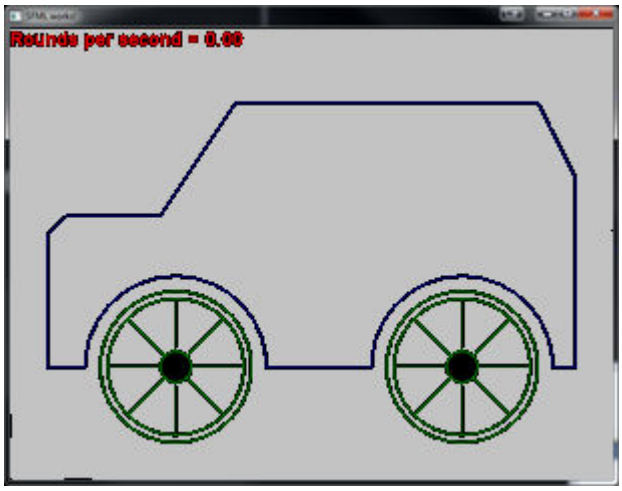
II. HLAModule

The implementation of the federate ambassador class `HLAModule` follows the steps discussed in the 2.2.3 Federate Development section.

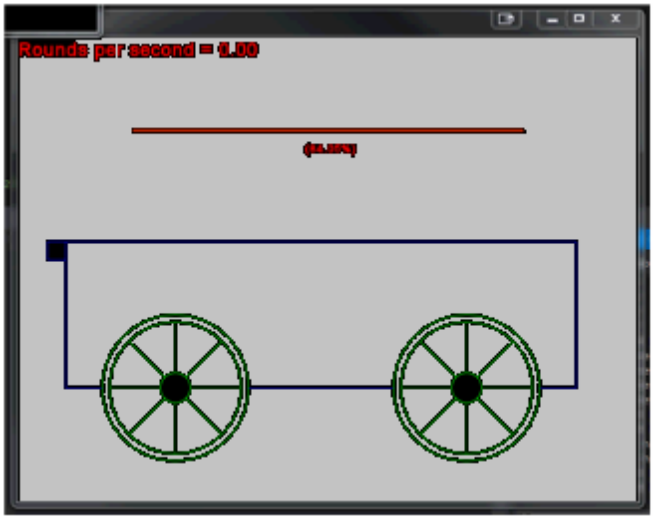
III. RTI

This implementation used an RTI provided by Pitch Technologies, pRTI Free [14]. Unfortunately, it is restricted to support only two federates. This is why the rope and load simulator had to be combined into one. Prototype 2 replaces this with another RTI from Portico project [15] that does not have this restriction.

Figure 3-4 below shows the graphical output of the HLA version of this prototype.



(a) Car



(b) RoadAndLoad

Figure 3-4: Prototype 1 (HLA) Graphical Output

3.3.3 Class Diagram

Figure 3-5 to 3-7 shows the class diagram of this prototype.

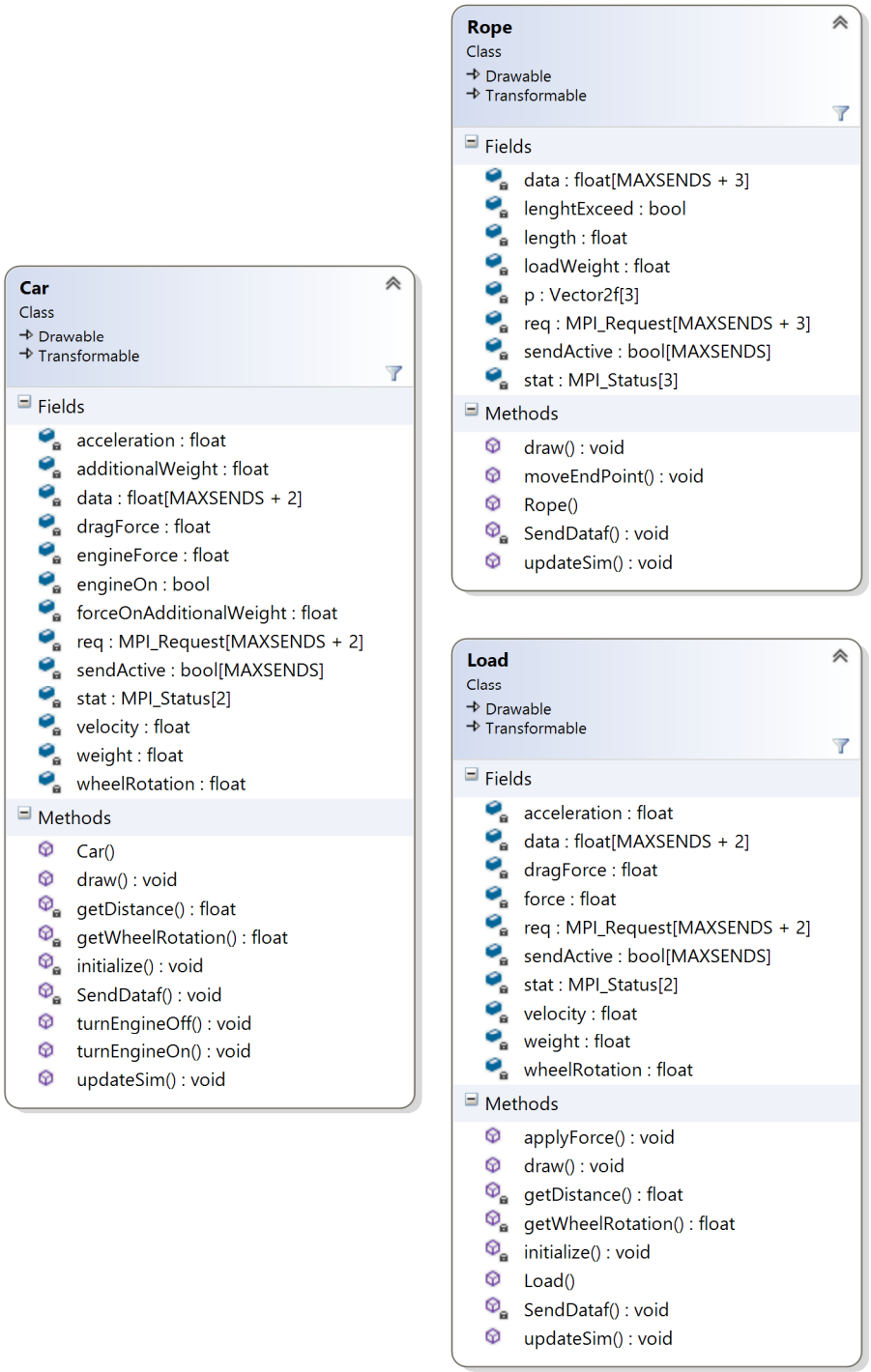


Figure 3-5: Prototype 1 Class Diagram - MPI Version

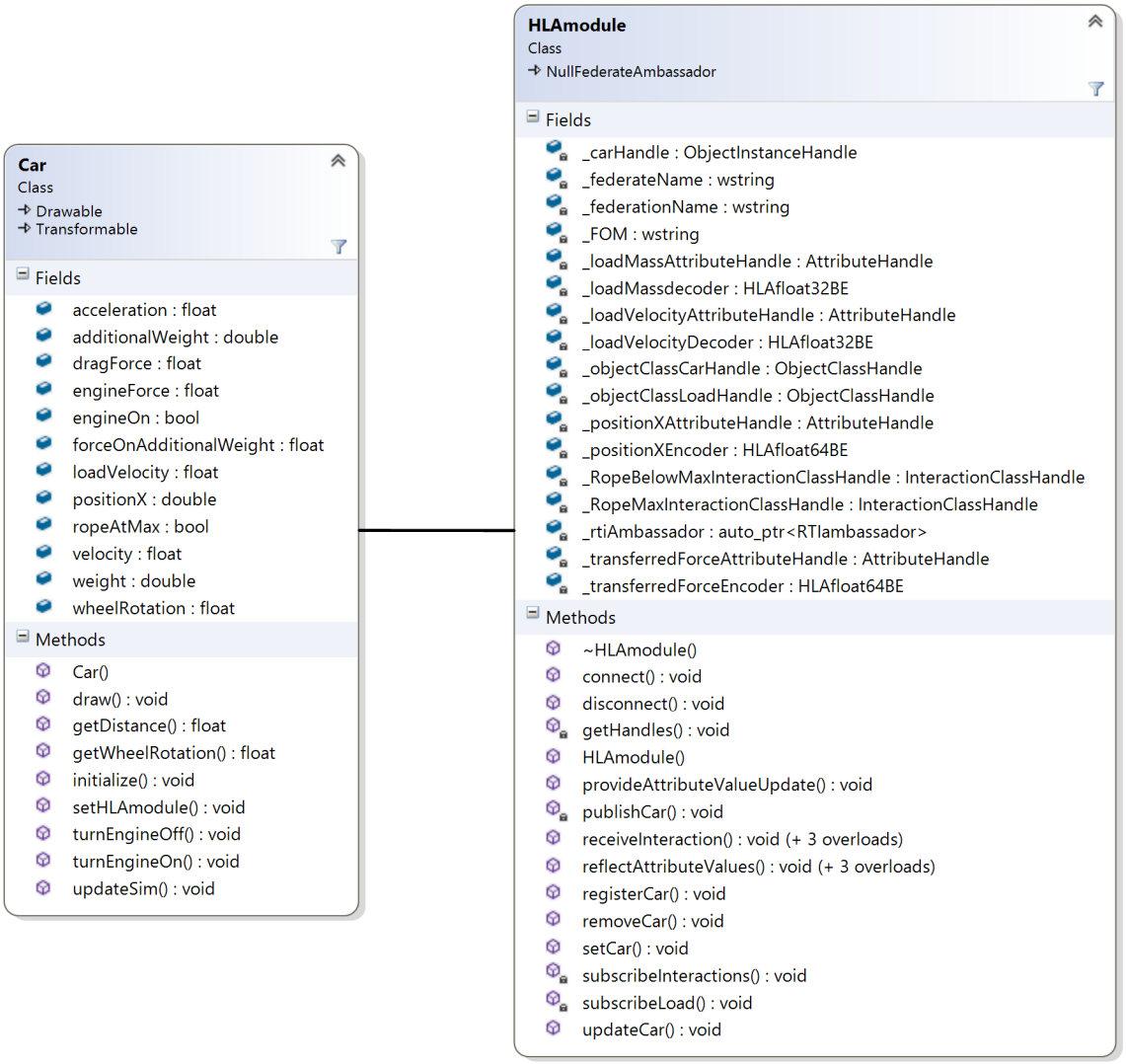


Figure 3-6: Prototype 1 Class Diagram - HLA Version - Car Federate

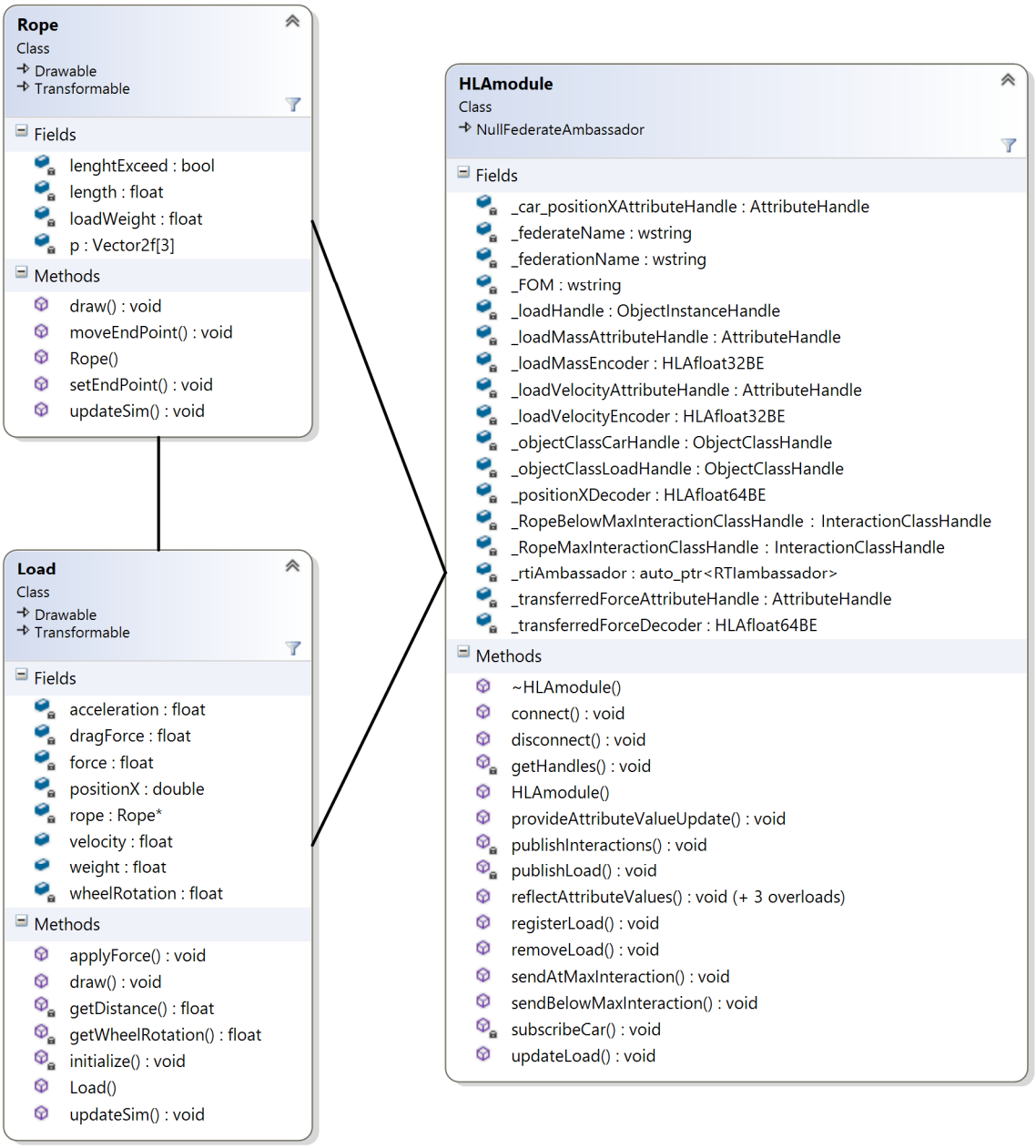


Figure 3-7: Prototype 1 Class Diagram - HLA Version - RoapAndLoad Federate

3.4 Observation

Initially, without the optimizations mentioned in the next section, the system was very jittery and quickly fell out of sync and frame rate ware below 30 frame per second. Messages were queuing up rapidly and were getting transferred late from when they were originated. This produced undesirable behaviors like the load acting as if the car is running even when it has stopped, the rope's length exceeding the maximum allowed length and rendering FPS of the scenes falling below real-time requirements of 30 frames per second. After the optimizations, the system could maintain process to process sync and real-time responsiveness, although the rope showed oscillation effect at the maximum length point. This is because the car kept advancing while the force information could get transferred to the load, and the load could update its status and catch up. A more powerful PC, real networked deployment, smaller calculation / time step could improve the oscillation, but improvement in time synchronization is needed to suppress this totally.

3.4.1 Optimization

The initial execution of the system made it apparent that further optimization had to be done to achieve acceptable performance. The first optimization was to reduce the frame rate. Initially, all three processes were set to run at about 60 frames per second. This corresponded to a communication load of up to 180 attempts per second on the underlying communication layer. Reducing this to 30 frames per second per process made the situation much more manageable.

The second, and most effective, performance improvement came from optimizing the code to require less frequent communications. Initially, all events were allowed to pass, but careful changes in the code to enable only modifications in values to be transferred made a noticeable difference. For instance, instead of passing on the force value at every iteration, it was made to transfer only on a change. The receiving party always worked with the most-recent value. With these improvements the system maintained a 30 frame per second performance.

3.4.2 HLA vs MPI

Although the comparison between HLA and MPI has appeared in literature before [42], in light of this implementation, several differences between these two architectures that are in favor of HLA, have become apparent. While the implementations did not show any observable performance difference between the two architecture, listed below are three (3) important differences that are related to this research.

I. Separation of Communication & Application Layer :

Although it is possible to design MPI systems manually that separate these two layers, HLA provides this separation inherently. The FOM and the federate ambassador implementation (in this case, the `HLAModule` Class) force a partition between the communication logic and the application logic. This allows the reusability of the communication layer that the second prototype, discussed in the next chapter, takes advantage of. It re-uses the `HLAModule` and part of the FOM from the towing simulator with little modification.

II. Data Distribution

Where MPI communication is primarily direct and point-to-point, HLA supports data distribution. This means, in MPI, if several processes need certain information, the provider generally has to be aware of that and send the data accordingly. However, in HLA, the RTI provides advance data distribution, where the producer can just send the data to RTI, and the RTI will take care of the responsibility of transferring that data to all interested parties. This makes the development of the participating simulator abstracted from each other.

III. Time Management

Although not directly used in this research, HLA provided advanced time management functionality that makes it suitable for possible future improvements that can be done to this research.

3.5 Conclusion from Prototype 1

The knowledge gathered in developing this prototype is the basis for the next phase of the research. Primarily, it provides important understanding about two popular communication paradigms, the MPI and the HLA. After considering the differences between them, HLA was selected to be the suitable candidate to handle the communication needs of the second prototype. This also makes it possible to identify the reusable components of the multi-craft system, specifically the FOM and `HLAModule` class of the HLA implementation. In the next chapter, the development of prototype 2 is presented, which is directly related to the multi-craft problem.

Chapter 4: Prototype 2 : Multi-Craft Water Simulation

4.1 Objectives

Compared to the first prototype, prototype 2 was designed to be more sophisticated. It is directly related to Model Based Division (MBD) of the multi-craft problem in the ocean environment. Therefore, the main objective is to prove the applicability of model based division in solving the multi-craft problem. To do this, there should be a water simulator and multiple floating object simulators. Each simulator has individual computing resources and simulates only its corresponding model. Interactivity is achieved by inter-simulator communication, which they do through an HLA driven network. HLA was chosen to be the suitable communication architecture for the advantages it provides over MPI for this scenario, such as data distribution and subscription, and implicit interaction. The water simulator is the most multifaceted component having visually interactive water and provision to send and receive data to and from each floating object simulator. On the other hand, the floating objects only communicate with the water simulator. They were chosen to be 3-Dimensional boxes having simple simulation behavior, like changing orientation based on changing terrain; in this case the water surface. This simplification of the floating objects' logic does not reduce our target problem of having complex multi-crafts. This is because each object has its own computing resources that it will not share. In real-world implementation, their complexity would be taken care of by employing appropriate processing power.

To enable interactivity, changes in water surface should affect the floating objects and movement of the objects should affect the water, such as creating ripples and wakes. There should be observable indirect interactions between multiple floating objects that happen via the water. For example, water surface disturbance created by one floating object will affect other nearby floating objects.

Another objective of this prototype is to observe MBD system performance and study the effect of network load on simulation throughput. The final objective is to design, identify and implement system components of a general MBD system.

The full source code of this prototype is included in the Supplementary Files. Here, analysis of the system and its different components are discussed.

4.2 System Analysis

The main requirement is a system that provides real-time simulation of water with multiple objects having complex models. As a consequence of the decision to divide the multi-craft simulation into networked components, communication becomes a significant part of this technique. The data that need to be communicated can be divided into two parts: a) onetime initialization data and b) continuous runtime data. The initialization data are information regarding the shape, starting position and orientation of each floating object. Shape data only need to include the parts that directly interact with the water for the water simulator to work. Figure 4-1, schematically illustrates a simulation environment where a ship is floating on water.

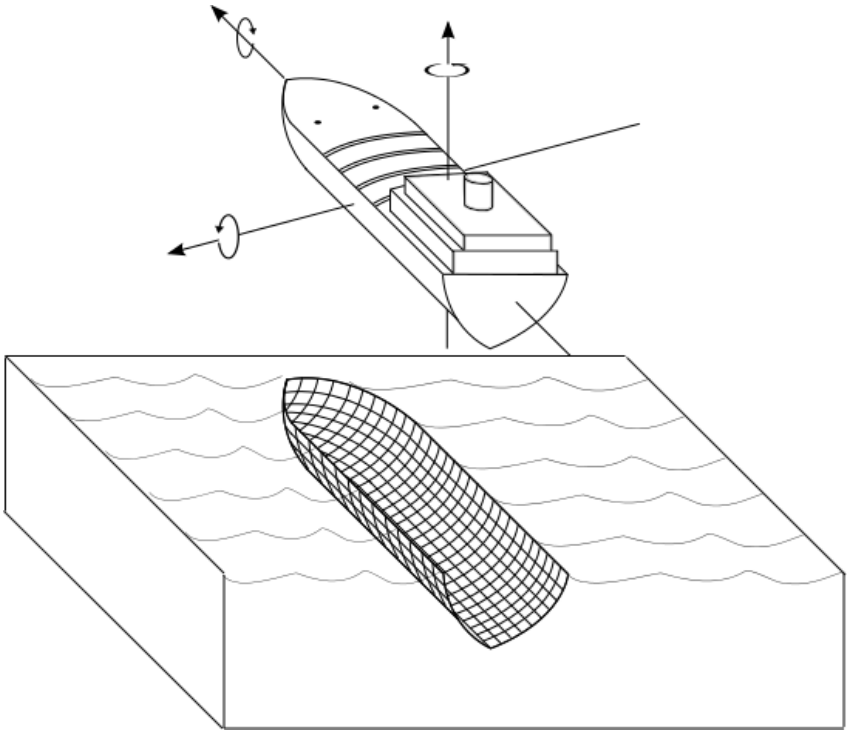


Figure 4-1: Water & Floating Object Simulator

The water simulator updates its wave heights in each update cycle in accordance with the algorithm that it uses. An appropriate algorithm should take into account the physical boundary of the water and all objects floating on it, where moving objects will create wake and disturbances on the water surface. Only the newly updated wave heights surrounding the floating object are communicated. In figure 4-1, it is shown as the hull shaped grid. Other information may also be communicated, like the logical time that has passed between two updates. Upon receiving these data, the floating object simulator, according to its simulation logic, will calculate its new states. For the ship in figure 4-1, among these new states are the new position and orientation data, namely surge, sway, heave, pitch, roll and yaw. These are shown as arrows and will be sent back to the water simulator so that the next update cycle can begin.

4.2.1 Network Load

The amount of data that needs to be communicated depends on the shape of the floating objects. This prototype considers the floating objects as square-shaped boxes. For one floating object in this prototype, the equivalent of the hull shaped grid in figure 4-1 is a 20x20 grid of wave heights amounting to a total of about 9.4 kilobytes of data. To be a viable solution for the types of scenarios that we want to handle as part of the multi-craft problem, the simulation needs to achieve at least 30 updates per second. That is a total of 282 kilobytes of data per second for each floating object that the water simulator needs to communicate in order to achieve an acceptable frame-rate.

4.2.2 States and Information Flow

Figure 4-2 illustrates a high level state diagram of the system. Each participating simulator goes through these transitions.

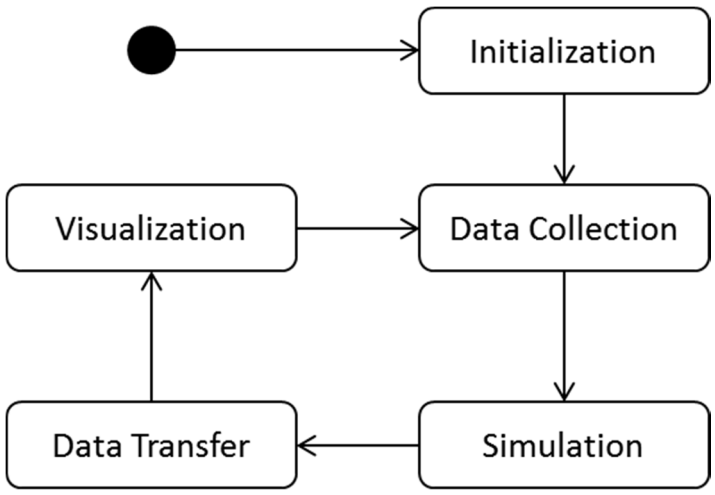


Figure 4-2: Prototype 2 - State Diagram

In the Initialization state, one-time communications occur, such as transferring the floating objects' shape information to the water simulator. After that, the main simulation loop starts. Each cycle begins in the data collection state by collecting new information; for example, updated wave heights from the water simulator and updated positions from the floating object simulators. Depending on the model, the simulation state advances the simulation in logical time. The resulting data is then transferred to all interested participants in the Data Transfer state. The cycle ends in the Visualization state where these updated data are presented for analysis, and a new cycle begins. Due to the implicit nature of the interaction inherent in the HLA communication infrastructure, all data collection and transfer happens in an asynchronous (non-blocking) manner.

Figure 4-3 shows the conceptual communication diagram of (a) the water simulator and (b) the floating object simulator. In both cases, the `Visualizer` collects the latest available information and draws appropriately to a window to show the current state of the simulator. In the water simulator, the `IWaveAlgorithm` class does the main work of using the position and orientation information from multiple `RemoteFloat` objects to update the wave height. The `FloatObject` class is the equivalent for the floating object simulator, which uses the `LocalWaveGrid` information to simulate that corresponding floating object.

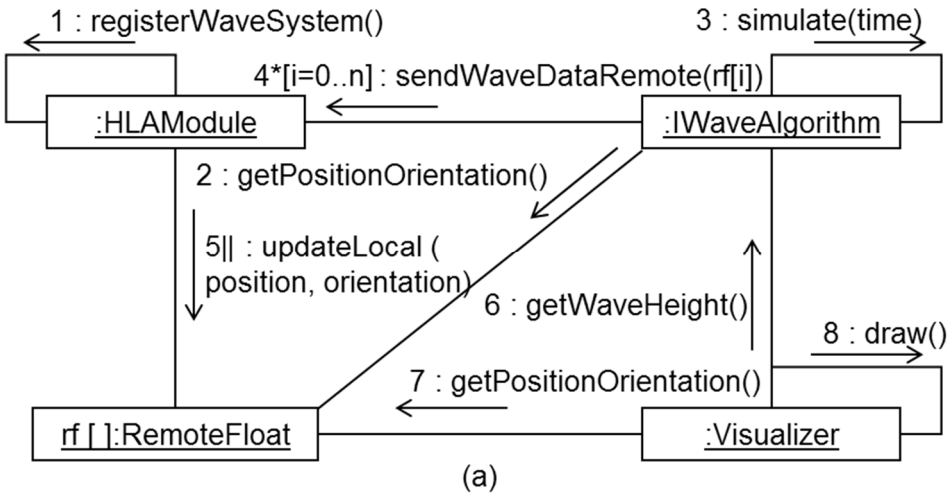


Figure 4-3: Prototype 2 - Communication Diagram a) Water Simulator, b) Floating Object Simulator

All updated information is sent to the `HLAModule` class for communication. The `HLAModule` class acts as an intermediary between the HLA Run Time Infrastructure (RTI) and the application. Among its responsibilities are registering to the RTI to establish what kind of data it wants to publish and subscribe to, and send and receive those data to and from the RTI.

4.3 System Components

The implementation of the prototype was done using C++ as the primary source language. Information on the main components of the system is presented below.

4.3.1 Federation Object Model

The FOM of this system primarily defines two (2) object classes. Table 4-1 lists these object classes, along with their attributes, data type of each attribute, update condition and publisher / subscriber information.

Table 4-1: Prototype 2 - Publish-Subscribe Matrix

Object Class	Attributes	Data Type	Update Type	Publisher	Subscriber
WaveSystem	WaveGrid	WaveGridType	On Change	WaveSystem	Floating Object
FloatingObject	PositionX	HLAfloat64LE	On Change	Floating Object	WaveSystem
FloatingObject	PositionY	HLAfloat64LE	On Change	Floating Object	WaveSystem

All floating object data is published by the floating object simulators and subscribed by the water simulator. `WaveGrid` is updated by the publishing water simulator and received by all the subscribing floating object simulators. The `Update Type` of `On Change` tells the RTI to communicate the values of the attributes whenever the value changes. `HLAfloat64LE`, used for the `PositionX` and `PositionY` attributes, is a 64-bit predefined floating-point data type. `WaveGridType` is a custom

data type that is based on arrays of `HLAfloat32LE` (32-bit float) elements. The cardinality of this user defined data type is dynamic.

4.3.2 HLAModule

The implementation of the `HLAModule` follows the steps discussed in the 2.2.3 Federate Development section. Due to the reusability inherent in a HLA system, the `HLAModule` developed in the first prototype was used here with little modifications. In addition, for receiving data from RTI this `HLAModule` makes use of the HLA provided encode helper classes, e.g., `HLAfloat64LE`, `HLAfloat64BE`, inside the `reflectAttributeValues()` function. These helper classes permit conversion of incoming data, formatted in little endian (LE) and big endian (BE) convention, to C++ data types. HLA provides helper classes for all common data types, such as integer, double, byte, and all common sizes, such as 16, 32, 64-bits.

4.3.3 Run Time Infrastructure

The initial development of the prototype used the free RTI provided by Pitch Technologies, pRTI TM Free. Although an efficient RTI, it is restricted to support at most two federates, a limitation that their commercial version does not have. That only allowed one floating object simulator and one water simulator. To alleviate this restriction, at later stages of development, the pRTI TM Free was replaced with the open-source Portico RTI. Portico RTI is free and does not have any restrictions. The replacement process is simple and straight forward, only requiring some environment variable and include-path

updates, and did not require any significant re-coding. This is due to the flexibility provided by the HLA specification that makes swapping RTI implementations easy.

4.3.4 Water Simulator

The water simulator implements the IWave algorithm detailed in section 2.3. The encapsulating class, `IWaveAlgorithm` is shown below, in figure 4-4.

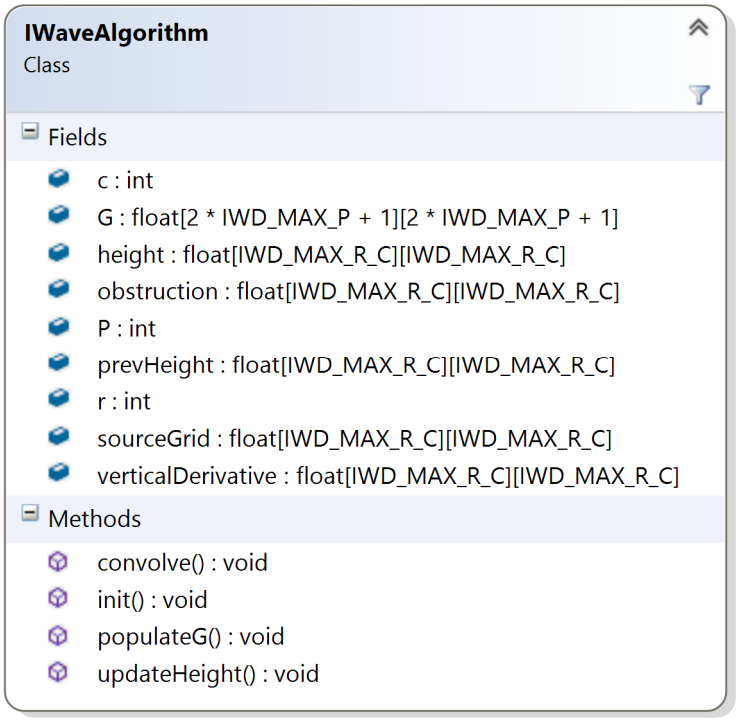


Figure 4-4: IWaveAlgorithm Class

The most important function is the public method,

```
void updateHeight(const double &dt)
```

It takes the time duration in the parameter `dt`, and updates the wave heights according to the IWave algorithm. Wave heights are stored in a 2-dimensional public

array `height[][]`, that the visualizer accesses to draw the graphical output. The energy source and obstructions are also 2-dimensional arrays named `sourceGrid[][]` and `obstruction[][]` respectively. The `RemoteFloat` class keeps these two arrays updated according to the position information received from the remote floating objects.

For simplification, a one-to-one correlation between simulated grid point and rendered grid point is kept. However, high-fidelity visualization could be achieved by having intermediate rendering grid points, and interpolating simulated wave heights.

4.3.5 Floating Object Simulator

The floating objects do not do any fluid simulation themselves. This simulator mainly does two (2) things: a) send position information to the water simulator through RTI and b) update the orientation of the local object according to the received wave heights. To send position updates, it has keyboard handling logic that takes movement commands from the user and calculates new positions. Orientation changes are simulated by calculating the overall angle of the local water surface underneath the object, and applying a rotation transformation to its geometry, which is a 3-dimensional box in this case.

For the current research, this is enough to visualize that the water surface changes are affecting the floating objects, and indirect interactions are occurring between multiple floating objects. However, a full-fledged floating object simulator, with an accurate model that takes into account gravity, drag, buoyancy and other hydrodynamic phenomenon to produce realistic behavior, could also be used, as long as the water federate and the floating object federate agree on a common FOM.

4.3.6 Graphics

The graphical output is generated using OpenGL version 3.3 and its corresponding shading language GLSL version 3.3. One important responsibility of the graphics subsystem is to prepare the wave heights for drawing. This is because the output of the IWave algorithm is a 2-dimensional grid of heights, but in OpenGL, 3-dimensional geometry is descretized using triangles.

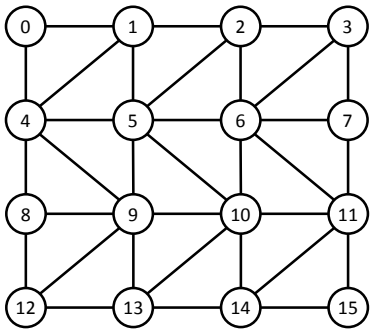


Figure 4-5: 2-D Wave Grid to Triangle Strip Conversion

A triangle is defined by an ordered list of three (3) vertices. Their order (clockwise / counter clockwise) will determine in which direction the corresponding triangle is facing. The `Visualizer::calculateTriangleStripIndices()` is a function designed specifically to calculate the order of indices of the 2-D grid that accurately converts it into a single triangle strip [43]. For the grid in figure 4-5, the output of the function would be 0, 4, 1, 5, 2, 6, 3, 7, 7, 11, 6, 10, 5, 9, 4, 8, 8, 12, 9, 13, 10, 14, 11, 15. This algorithm is better than having to calculate one strip for each row, because the whole grid can be rendered by a single OpenGL draw call. Moreover, since only the height of each point changes and not the position on the 2-D grid, this calculation to generate the order of indices is done once at initialization, and stored to be reused for all subsequent draw calls.

Two important components of the graphics subsystem are the camera and light model. The implemented light system could simulate ambient, diffuse, and specular lights. The camera offered smooth mouse, and keyboard guided movements that allowed observing the simulation from multiple perspectives.

Figure 4-6 shows the graphical output of the water simulator accommodating two remotely connected floating boxes. The flattened red and green areas represent two remote floating objects. Those could be controlled in the remote machines to move in any direction, causing their changed position to be transmitted to the water simulator via the HLA RTI. The figure shows the deformed water surface, generated by the IWave logic, as a result of these remote movements. The water simulator was processing an average 80 frames per second, on a moderately powerful computer running a 2.4 GHz Pentium Processor with eight (8) Gigabytes of memory.

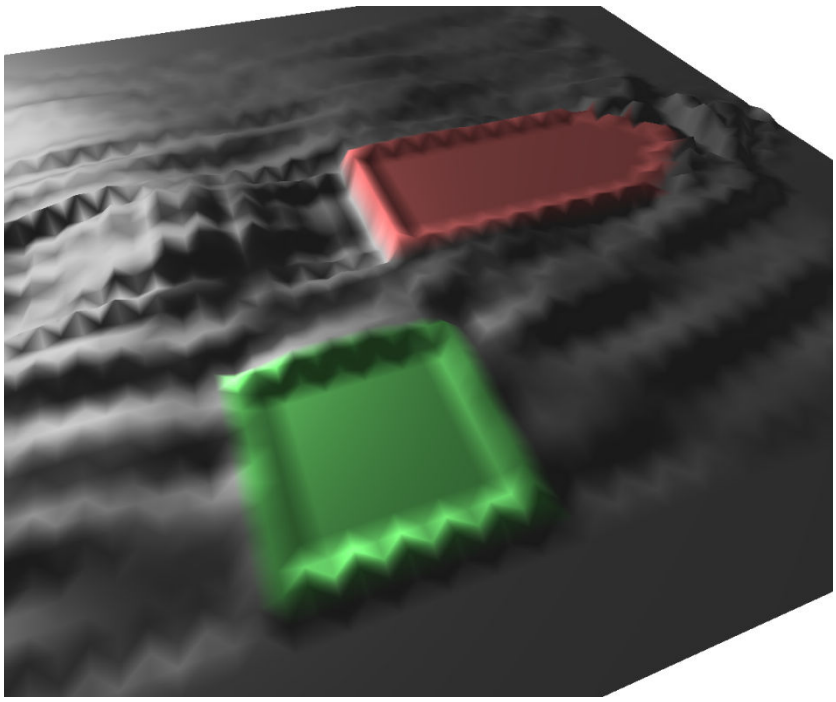


Figure 4-6: Water Simulator Output

4.3.7 Class Diagram

Figure 4-7 shows the class diagram of prototype 2.

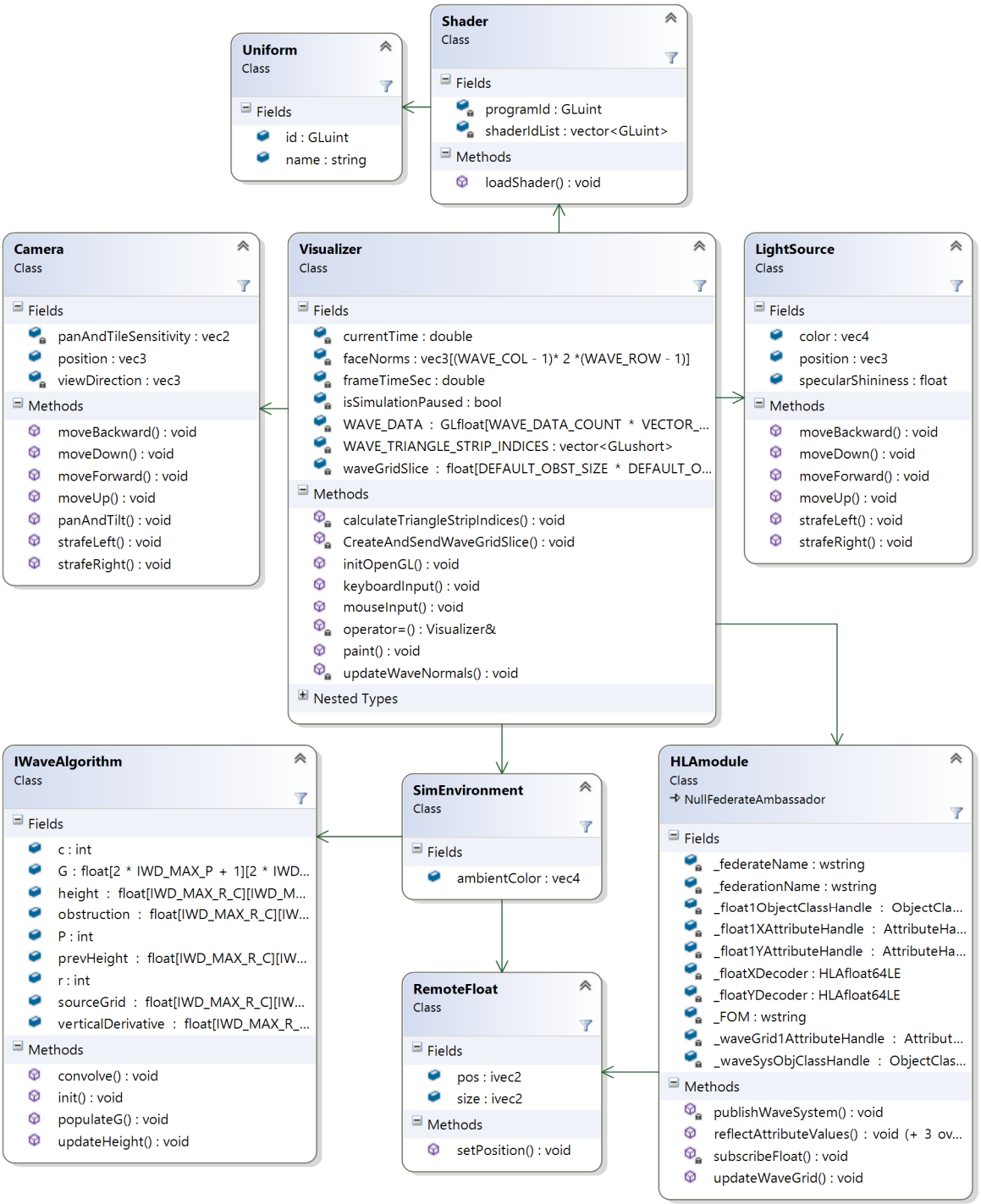


Figure 4-7: Prototype 2 - Class Diagram

4.4 Results of Communication Load Test

As the water simulator is the one communicating with all objects, whereas the floating objects communicate only with the water simulator, a number of tests were conducted to evaluate the communication capacity of the water simulator. Figure 4-8 shows the frame/second performance of the water simulator with increasing communication load for two different setups. The computers used were moderately powerful running a 2.4 GHz Pentium Processor with eight (8) Gigabytes of memory. For case 1, the experimental setup consisted of a single machine running the water and floating object federates on the same processor. On the other hand, case 2 shows the results where all floating objects were simulated on one computer and the water simulator ran on a separate computer connected via a Wireless Area Network (WAN). In both cases, the water simulator had full functionality, but due to the number of floating objects involved, a simplified implementation of them was used with reduced simulation logic that primarily communicated data with the RTI. For the single computer run, the floating object's visualization had to be turned off, because having both the water and floating object federates accessing the sole graphics processing unit made the performance drop significantly.

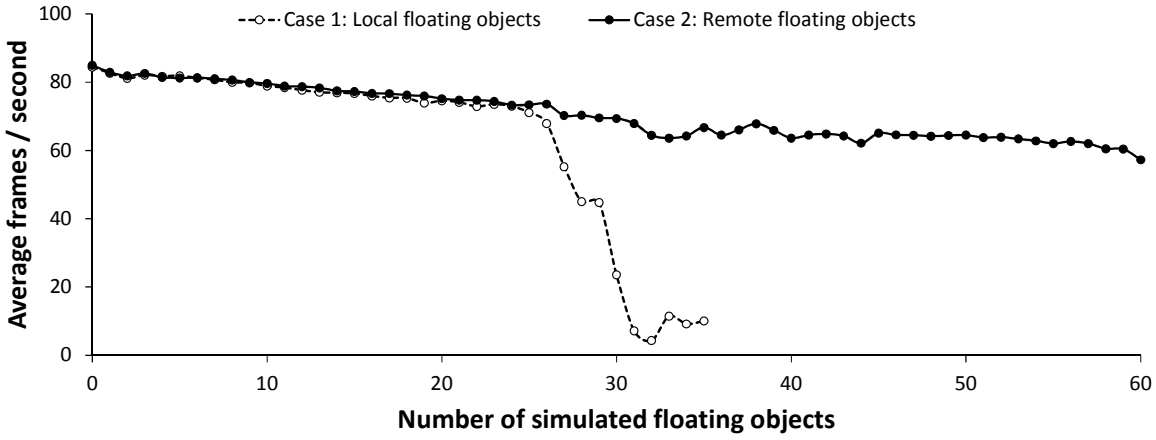


Figure 4-8: Communication Load Test

Although these performance measures are dependent on multiple aspects, such as the power of the system running the water simulator, the size of the floating objects and the efficiency of the algorithm, for this prototype, it was observed that the limit of floating objects was about 25 for case 1. Beyond that the system became unstable, and the time gap between frames became inconsistent.

Interestingly, for the remote machine test, the water simulator was more stable, having a consistent, low frame drop for more than twice the amount of communication load and still remained above 60 frames/second. The test was terminated at about 60 floating objects, because the remote machine hosting the floating objects started to struggle to process all these data coming from the water simulator and crashed. It is possible that if the remote machine had more processing power, or if there were multiple remote nodes, then the water simulator could have handled even more load.

The better performance of the networked setup is attributable to not having to timeshare the CPU between multiple competing federates compared to the single machine setup. The networked setup also allows all visualizations to be turned on.

4.5 Conclusion

This prototype acts as a proof of concept that, for the multi-craft simulation, Model Based Division using HLA is a well-suited solution. It shows how the multi-craft problem can be divided, implemented and deployed to achieve two-way water-object interactivity without sacrificing complexity or accuracy. The development details give insights into the different components of the system, their interactions and the overall communication requirement. The capability of the communication layer, as evident from the communication load test, suggests that, in a production grade deployment this system should be capable of simulating real-world multi-craft scenarios, such as a lifeboat escorted by ice-breaker, ice field clearing using wake wash. From the load test, we also see the quickly diminishing return of a single system solution, where even without visualization the system failed to achieve the level of performance of the networked setup.

Chapter 5: Conclusion

The magnitude of the multi-craft problem prohibits the extent of solutions that can be optimized and implemented in a single-system environment. This prohibition results in compromises in realizing a full solution for the multi-craft problem by reducing or simplifying pieces of the puzzle, such as sacrificing accuracy in the water simulation, reducing the complexity of the floating objects or restricting the maximum number of floating objects permitted in a scenario. The current study presents a distributed solution for the multi-craft problem that has the potential to overcome these limitations and allows the water and floating object simulations to retain as much computational complexity as required.

The two prototype implementations serve as a proof of concept and provide evidence for the applicability of the solution to this problem. The first prototype provides valuable insights into MPI and HLA and helps the decision of choosing HLA for the advanced implementation later. The second prototype shows the applicability of MBD in solving the multi-craft problem and demonstrates, through communication load testing, the possibility of having several floating objects connected remotely. They also answer many design questions, such as the data requirements and information flow.

5.1 Contribution

The most important contribution of this study is in devising a suitable solution for the multi-craft problem. The research shows that there are primarily five (5) areas where the model-based division approach excels :

I. Modularity

Increased modularity is a direct benefit of this technique of determining the scope of each simulator by its model. This allows all participating simulators to be designed independently of each other by only agreeing on their published and subscribed data. Moreover, the HLAModule is capable of abstracting the encoding of the data from the application layer. This makes implementation and debugging easier because it is harder for errors to propagate from one simulator to another.

II. Possibility of Individual Optimization

The multi-craft simulation is inherently parallel and suitable for distributed systems. Since each floating object is separated in programming logic, they can be simulated in parallel. This solution enables object-water and indirect object-object interactions to be achieved as described. Further, it frees the floating object simulator from having to perform any water-related calculation. The modularity of this method thus simplifies optimization by permitting separate performance tuning of each simulator without considering the system as a whole.

III. Scalability

This system is scalable because introducing a new floating object does not complicate the water simulator. The majority of the new object simulation load is taken care of by a remote process and only the water simulator has to deal with a new shape and added data communication. Compared to a single machine system, this is more scalable and is only limited by the capacity and latency of the network.

IV. Reliability

This technique also grants enhanced system stability in two major ways. One is by separation, which inhibits error propagation, and also by allowing indirect interaction through HLA, which makes the system more resilient to failure by allowing any participating simulator to crash or stop without shutting the whole simulation down.

V. Diversity of Floating Object Models

The indirect interaction provided by HLA also has the possible benefit of allowing variability in floating object models. A high fidelity floating object, which requires multiple kinds of information from the water simulator, i.e. height field, velocity field, and pressure field, can co-exist with a low fidelity floating object, requiring only a small subset of all available information. In this case, different floating objects subscribe to different sets of data, and the RTI manages appropriate data delivery on runtime.

5.2 Future Recommendation

This study opens up a couple of future research prospects. Here are two (2) main areas where this research can be expanded,

a) Large-Scale Implementation:

All the testing done for this research involves at most two computers. Large-scale testing in a production grade environment involving several machines simulating accurate fluid and floating objects would generate valuable experimental data to further validate the significance of this solution. This would make it possible to take advanced performance measurements and analyze the effects of the design on optimization and

scalability. It would also allow those complex multi-craft scenarios to be examined that acted as the motivation behind this research, such as an ice-breaking ship leading a supply ship or a lifeboat, effects of ship wake wash on an ice-field.

b) Other Fluid Algorithm:

Study of the adaptability of this technique with different known interactive fluid simulation algorithms can also be a worthwhile addition to this research.

Furthermore, to accommodate the most complex of models, improvements will have to be made to the design, such as inclusion of a more advanced time synchronization scheme and network data compression.

Bibliography

[1] D. Enright, S. Marschner and R. Fedkiw. “Animation and rendering of complex water surfaces,” *ACM Transactions on Graphics (TOG)*, vol. 21(3), pp. 736-744, July 2002.

[2] N. Foster and R. Fedkiw. “Practical animation of liquids,” in *Proc. 28th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 23-30, 2001.

[3] H. Yan, Z. Wang, J. He, X. Chen, C. Wang and Q. Peng. “Real-time fluid simulation with adaptive SPH,” *Computer Animation and Virtual Worlds*, vol. *20(2-3),* pp. 417-426, June 2009.

[4] M. Carlson, P. J. Mucha and G. Turk. “Rigid fluid: Animating the interplay between rigid bodies and fluid,” *ACM Transactions on Graphics (TOG)*, vol. 23(3), pp. 377-384, August 2004.

[5] S. Clavet, P. Beaudoin and P. Poulin. “Particle-based viscoelastic fluid simulation,” in *Proc. 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 219-228, 2005.

[6] Y. Liu, X. Liu and E. Wu. “Real-time 3D fluid simulation on GPU with complex obstacles,” in *Proc. Computer Graphics and Applications, 12th Pacific Conference*, pp. 247-256, 2004.

[7] T. Takahashi, H. Ueki, A. Kunimatsu and H. Fujii. “The simulation of fluid-rigid body interaction,” in *Proc. ACM SIGGRAPH 2002 Conference Abstracts and Applications*, pp. 266, 2002.

[8] S. Ueng, D. Lin and C. Liu. “A ship motion simulation system,” *Virtual Reality*, vol. 12(1), pp. 65-76, March 2008.

[9] “Message Passing Interface Forum,” Internet: http://www.mpi-forum.org/, [February 16, 2013].

[10] Microsoft corporation. “HPC pack 2012 MS-MPI,” Internet: http://www.microsoft.com/en-us/download/details.aspx?id=36045, November 12, 2012 [February 20, 2013].

[11] B. Barney and L. Livermore. “Message passing interface (MPI),” Internet: https://computing.llnl.gov/tutorials/mpi/, June 17, 2015 [February 16, 2013].

[12] HLA Working Group and others. “IEEE standard for modeling and simulation (M&S) high level architecture (HLA) - framework and rules,” *IEEE Std.1516-2000* 2000. DOI: 10.1109/IEEESTD.2000.92296.

[13] T. Lu, C. Lee, W. Hsia and M. Lin. “Supporting large-scale distributed simulation using HLA,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 10(3), pp. 268-294. July 2000.

[14] Pitch Technologies. “Pitch pRTI Free,” Internet: http://www.pitch.se/downloads/pitch-prti-free, [August 3, 2013].

[15] “The Portico Project,” Internet: http://www.porticoproject.org, [March 4, 2014].

[16] HLA Working Group and others. “IEEE standard for modeling and simulation [M&S] high level architecture (HLA) - federate interface specification,” *IEEE Std 1516.1-2000* 2001. DOI: 10.1109/IEEESTD.2001.92421.

[17] HLA Working Group and others. “IEEE standard for modeling and simulation (M&S) high level architecture (HLA) - object model template (OMT) specification,” *IEEE Std 1516.2-2000* 2001. DOI: 10.1109/IEEESTD.2001.92423.

[18] J. Tessendorf. “Interactive water surfaces,” in *Game Programming Gems 4*, vol. 4, Charles River Media, 2004, pp. 265-274.

[19] Mastin, Gary and Watterberg, Peter and Mareda, John F and others. “Fourier synthesis of ocean scenes,” *IEEE Computer Graphics and Applications*, vol. 7(3), pp. 16-23, March 1987.

[20] E. Darles, B. Crespin, D. Ghazanfarpour and J. Gonzato. “A survey of ocean simulation and rendering techniques in computer graphics,” *Computer Graphics Forum*, vol. 30(1), pp. 43-60, 2011.

[21] J. Tan and X. Yang. “Physically-based fluid animation: A survey,” *Science in China Series F: Information Sciences*, vol. 52(5), pp. 723-740, 2009.

[22] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn and T. J. Purcell. “A survey of general-purpose computation on graphics hardware,” *Computer Graphics Forum*, vol. 26(1), pp. 80-113, 2007.

[23] A. R. Brodtkorb, T. R. Hagen and M. L. Sætra. “Graphics processing unit (GPU) programming strategies and trends in GPU computing,” *Journal of Parallel and Distributed Computing,* vol. 73(1), pp. 4-13, January 2013.

[24] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips. “GPU computing,” in *Proc. IEEE*, vol. 96(5), pp. 879-899, May 2008.

[25] C. L. Fefferman. “Existence and smoothness of the navier-stokes equation,” *The Millennium Prize Problems*, pp. 57-67, 2000.

[26] J. J. Monaghan. “Smoothed particle hydrodynamics,” *Reports on Progress in Physics*, vol. 68(8), pp. 1703, 2005.

[27] E. Wu, Y. Liu and X. Liu. “An improved study of real-time fluid simulation on GPU,” *Computer Animation and Virtual Worlds*, vol. 15(3-4), pp. 139-146, 2004.

[28] F. Zhang, L. Hu, J. Wu and X. Shen. “A SPH-based method for interactive fluids simulation on the multi-GPU,” in *Proc. 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, pp. 423-426, 2011.

[29] S. Chen and G. D. Doolen. “Lattice boltzmann method for fluid flows,” *Annual Review of Fluid Mechanics*, vol. 30(1), pp. 329-364, 1998.

[30] P. R. Rinaldi, E. A. Dari, M. J. Vénere and A. Clausse. “A lattice-boltzmann solver for 3D fluid simulation on GPU,” *Simulation Modelling Practice and Theory*, vol. 25, pp. 163-171, June 2012.

[31] F. Kuznik, C. Obrecht, G. Rusaouen and J. Roux. “LBM based flow simulation using GPU computing processor,” *Computers & Mathematics with Applications*, vol. 59(7), pp. 2380-2392, 2010.

[32] J. Ricardo da Silva Junior, E. W. Gonzalez Clua, A. Montenegro, M. Lage, M. d. A. Dreux, M. Joselli, P. A. Pagliosa and C. L. Kuryla. “A heterogeneous system based on GPU and multi-core CPU for real-time fluid and rigid body simulation,” *International Journal of Computational Fluid Dynamics*, vol. 26(3), pp. 193-204, 2012.

[33] C. Everitt. “Interactive order-independent transparency,” *White Paper, nVIDIA*, vol. 2(6), pp. 7, 2001.

[34] C. Braley and A. Sandu. “Fluid simulation for computer graphics: A tutorial in grid based and particle based methods,” *Virginia Tech, Blacksburg,* 2010.

[35] J. M. Cohen, S. Tariq and S. Green. “Interactive fluid-particle simulation using translating eulerian grids,” in *Proc. 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 15-22, 2010.

[36] N. Chentanez and M. Müller. “Real-time eulerian water simulation using a restricted tall cell grid,” *ACM Transactions on Graphics (TOG)*, vol 30(4), pp. 82:1-82:10, July 2011.

[37] C. Batty, F. Bertails and R. Bridson. “A fast variational framework for accurate solid-fluid coupling,” *ACM Transactions on Graphics (TOG*), vol. 26(3), pp. 100, July 2007.

[38] R. Lubbad and S. Løset. “A numerical model for real-time simulation of ship-ice interaction,” *Cold Regions Science and Technology*, vol. 65(2), pp. 111-127, 2011.

[39] K. McTaggart and R. Langlois. “Physics-based modelling of ship replenishment at sea using distributed simulation,” in *Proc. SNAME Annual Meeting and Expo*, vol. 1040, 20-23 October 2009.

[40] K. de Kraker, J. Duncan, E. Budde and R. Reading. “NATO standards for virtual ships,” in *Proc. Fall 2005 Simulation Interoperability Workshop*, Paper 05F-SIW-020, 2005.

[41] T. Bastin, A. Zubayer, B. Veitch, A. Akinturk, J. Wang and R. Billard. “Propeller wake wash for ice management,” in *Proc. Oceans-St. John's, 2014*, pp. 1-5, 14-19 Sept. 2014.

[42] K. Rycerz, A. Tirado-Ramos, A. Gualandris, S. F. P. Zwart, M. Bubak and P. M. Sloot. “Interactive N-body simulations on the grid: HLA versus MPI,” *International Journal of High Performance Computing Applications*, vol. 21(2), pp. 210-221, 2007.

[43] D. Lecocq. “Triangle strip for grids – A construction,” Internet: http://dan.lecocq.us/wordpress/2009/12/25/triangle-strip-for-grids-a-construction/, December 25, 2009 [January 10, 2014].