Design, Implementation, and Performance Evaluation of HLA in Unity

Author: Karl Söderbäck

2017-06-09

Supervisor: Alachew Mengist
Examiner: Peter Fritzson
Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Upphovsrätt

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Karl Söderbäck
Abstract

This report investigates if an HLA-plugin for the game engine Unity can be made and whether or not it would lead to any drawbacks in regard to data exchange and performance. An implementation of a plugin and performance tests on it proceeds to show that the possibilities of running HLA as a plugin in Unity shows a lot of promise for 3D-applications designed in Unity communicating over HLA.

Keywords: HLA, Unity, plugin, game engine, simulation, distributed simulation, performance.
Acknowledgement

I would like to thank Peter Fritzon, my examiner at Linköpings University and my supervisor Alachew Mengist for providing support and guidance in my thesis report. I also want to thank Pitch Technologies and my supervisor at the company, Thomas Brännström for helping me learn HLA and discussing solutions along the way.
Table of Contents

Abstract ........................................................................................................................... 3
Acknowledgement ........................................................................................................... 4
Table of Contents ........................................................................................................... 5
List of Figures ................................................................................................................. 7
List of Tables .................................................................................................................. 8
List of Listings ................................................................................................................. 9
Acronyms ....................................................................................................................... 10

1. Introduction .................................................................................................................. 11
   1.1 Motivation ................................................................................................................ 11
   1.2 Aim .......................................................................................................................... 11
   1.3 Research Questions ............................................................................................... 11
   1.4 Limitations ............................................................................................................. 12
   1.5 Thesis Outline ....................................................................................................... 12

2. Theoretical Background ............................................................................................... 13
   2.1 Unity ....................................................................................................................... 13
   2.1.1 Managed Plugins ............................................................................................. 13
   2.1.2 Native Plugins .................................................................................................. 13
   2.2 High Level Architecture (HLA) ............................................................................ 14
   2.2.1 HLA Object Models ....................................................................................... 16
   2.2.2 HLA Rules ....................................................................................................... 16
   2.2.3 Performance in HLA ...................................................................................... 17
   2.3 Tools and Environment ......................................................................................... 17
   2.3.1 Pitch Developer Studio ................................................................................ 17
   2.3.2 Pitch pRTI ....................................................................................................... 17
   2.3.2 The Tank Simulation ........................................................................................ 18

3. Design and Evaluation Method .................................................................................... 20
   3.1 HLA-plugin in Unity .............................................................................................. 20
   3.2 Performance Evaluation ....................................................................................... 21
   3.2.1 Frame Rate Testing ......................................................................................... 22
   3.2.2 RTT Testing ...................................................................................................... 22

4. Implementation and Test Results ................................................................................. 23
   4.1 HLA-plugin in Unity .............................................................................................. 23
   4.2 Performance Evaluation ....................................................................................... 25
   4.2.1 Frame Rate ....................................................................................................... 26
   4.2.2 RTT ................................................................................................................... 26

5. Discussion ................................................................................................................... 28
   5.1 HLA-plugin in Unity .............................................................................................. 28
List of Figures

Figure 2.1. A federation with two federates ........................................................................................................ 15
Figure 2.2. Pich pRTI displaying information about the federation TankGame with federates DamageServer, Tank and Tank-1................................................................................................................... 18
Figure 2.3. Two tanks facing each other in the tank simulation. ........................................................................ 19
Figure 3.1 The design and implementation process of the HLA-plugin .......................................................... 20
Figure 3.2. A simple sketch of the testing setup ................................................................................................ 21
Figure 4.1. The tank simulation from Figure 6 visualized in 3D using the HLA-plugin in Unity, displayed from the point of view of Tank95 .................................................................................................................. 25
List of Tables

Table 1. Frame rate after creating tanks using HLA ......................................................... 26
Table 2. Frame rate after creating tanks locally in Unity .................................................. 26
Table 3. Time measurements ............................................................................................. 27
List of Listings

Listing 2.1. Defining __declspec(dllexport) as DLLExport for future calls ........................................... 13
Listing 2.2. C++ functions within a C-wrapper to be exported as a Native plugin ................................. 14
Listing 2.3. The function called hlaconnect () imported from the DLL to Unity .................................. 14
Listing 4.1. The struct holding all necessary information about a tank ................................................. 23
Listing 4.3 The function get_last_joined is called with a StringBuilder, its capacity and a tankinfo struct
as parameters ........................................................................................................................................ 24
Listing 4.4 get_last_joined takes a StringBuilder as a char*, its capacity as an int and the tankinfo struct
as parameters ........................................................................................................................................ 24
Acronyms

HLA – High Level Architecture
DLL – Dynamic-Link Library
RTI – RunTime Infrastructure
SOM – Simulation Object Model
FOM – Federation Object Model
MOM – Management Object Model
OMT – Object Model Template
1. Introduction

This thesis was proposed by Pitch Technologies in Linköping where the work has been conducted. This chapter presents the motivation, objective and limitations of the thesis.

1.1 Motivation

Pitch Technologies [1], further mentioned as Pitch, is a company based in Linköping which provides solutions and services for distributed simulations. Their work is highly focused on using the standard called High Level Architecture (HLA) [5]. Pitch has an interest in the possibility of running HLA as a plugin for the game engine Unity in order to create better visualizations of certain simulations.

HLA enables simulations running on different platforms of different kinds to interact with each other. HLA is a narrow field and few are familiar with the standard. To increase the knowledge of it, Pitch has training programs where they teach HLA. One part of this training is a simulation where users can control tanks in a two-dimensional environment and handle all communication using HLA.

To increase the quality of this training program, the company is interested in visualizing the two-dimensional tank simulation in three dimensions by running HLA as a plugin in Unity. Apart from enhancing the quality of the tank simulation, proving that HLA can be run as a plugin in Unity opens up a lot of new and interesting possibilities for other software used by Pitch. Since a lot of updates are sent frequently using HLA in order to make the simulations accurate, it is important that an HLA-plugin for Unity does not lose performance in order to be useful.

1.2 Aim

The goal of the thesis is to analyze the requirements for an HLA-plugin in Unity and find a satisfying implementation of a plugin. There are two main aspects to consider. The first aspect is how data should be exchanged between HLA and Unity. The second aspect is whether or not performance is lost by running HLA as a plugin for Unity. This is interesting as too much performance loss would lead to misleading visualizations of the simulations in Unity which would make the plugin irrelevant and useless.

1.3 Research Questions

The research questions of this thesis are based on the main aspects described in section 1.2. Considering those aspects, the following questions are relevant to answer:

- How does running HLA as a Unity-plugin affect the possibilities of data exchange in a simulation?
- Is performance lost when HLA is used as a plugin for Unity?
1.4 Limitations

Many Pitch clients use real world map data for their simulation. Considering that, the possibilities of generating terrain based on real world maps in Unity would be interesting to investigate. However, due to time restrictions of the thesis this issue is not investigated.

1.5 Thesis Outline

Relevant technologies and standards for the thesis are explained in Chapter 2. In Chapter 3, we present the method to implement the HLA plugin and answer the research question. Chapter 4 presents the results brought forth by the method and the discussion of the result is presented in Chapter 5. Conclusions of the thesis and future work are presented in Chapter 6.
2. Theoretical Background

This chapter explains relevant parts of Unity and the HLA standard to increase the understanding of what tools and techniques are available for the work. The tools used and the two-dimensional tank simulation that the work is based on are explained further in this chapter as well.

2.1 Unity

Unity is a game development tool used to design games in both 2D and 3D. There are four available versions of the engine: Personal, Plus, Pro and Enterprise. Personal is the free version which anyone can download and use and it contains features necessary for beginners working with Unity. Plus, and Pro contain more advanced features suitable for more serious game developers that have experience working with the engine. The Enterprise version is for larger organizations and is tailored to suit the needs of the organizations using it [3].

Usually all functionality within a Unity project is achieved by writing scripts. However, certain functionality can be fetched from code created outside of Unity. This is done using plugins. There are two kinds of plugins available, Managed plugins and Native Plugins [2].

2.1.1 Managed Plugins

Managed plugins contain .NET code and can only access features supported by the .NET libraries. Unity uses standard .NET tools to compile its scripts and the managed code is accessible to these tools, thus there is not a big difference between script code written in Unity and code from managed plugins. The biggest difference is that the managed code is compiled outside of Unity.

2.1.2 Native Plugins

Native plugins can be used to get functionality from code libraries completely unknown to Unity. Third party libraries which, for example, hold specific features like OS-calls or certain hardware specific calls can be accessed in Unity using native plugins. The downside of using native plugins is that Unity cannot access these libraries in the same way that it can with managed code. Problems with managed code can generate standard compiler errors while native plugins first give an error when the developer tries to run the project.

Native plugins are used by first writing code in a C-based language, like C++. This code is then compiled into a Dynamic-link Library (DLL) and available in a .dll file. In order to make functions in a DLL available when the .dll file is exported, they need to be marked for export using the __declspec(dllexport) command which is defined as DLLExport in Listing 2.1.

```
#define::DLLexport __declspec(dllexport)
```

Listing 2.1. Defining __declspec(dllexport) as DLLExport for future calls
In order to avoid name-mangling issues when compiling the C++ code, the functions that are going to be exported in the DLL to Unity needs to be surround by the C-wrapper (extern “C”) in the way Listing 2.2 illustrates.

```c
extern "C"
{
  DLExport void hlaconnect();
  DLExport void hla_disconnect();
  DLExport bool get_updated(char* lm, int length, tankinfo* ti);
  DLExport bool getTank_delete_boolean();
  DLExport bool get_fire_boolean();
  DLExport bool get_joined_boolean();
  DLExport void get_firecoordinates(firecoordinates* fire_co);
  DLExport int get_last_joined(char* l, int length, tankinfo* ti);
  DLExport void get_deleted_tank(char* del, int length);
}
```

*Listing 2.2. C++ functions within a C-wrapper to be exported as a Native plugin*

To access the plugin and the exported functions from Unity, the functions need to be declared the way that is illustrated in Listing 2.3.

```c
[DllImport("tankdll64")]
public static extern void hlaconnect();
```

*Listing 2.3. The function called hlaconnect () imported from the DLL to Unity*

As shown in Listing 2.3, the DllImport command takes the name of the .dll file which is found automatically if it is placed in folder with the name Plugins within the Unity project. The imported function hlaconnect () then needs to be declared as public static extern in order to be called from Unity [4].

### 2.2 High Level Architecture (HLA)

HLA is a technical architecture designed for interoperability in simulations. HLA is based on the idea that no simulation can satisfy all possible users or uses. A simulation designed for one specific use
could however complement another simulation thus making itself reusable. HLA provides the structure which makes it possible to have a set of simulations interact with each other disregarding what platform they are running on. This means that, for example, a simulation running only artificial intelligence components can interact with a flight simulator controlled by a human. A set of simulations interacting with each other in HLA is called a Federation. Every simulation within a federation is called a Federate.

HLA does not demand that a federate is represented in a specific way but all federates are required to have the necessary capabilities for interacting with objects in other simulations. The data exchange between federates in HLA is handled by the Runtime Infrastructure (RTI). The RTI handles all interactions in a federation by providing a set of services that are available to help federates interact. Figure 2.1 illustrates an HLA federation with two federates connected to the RTI.

![Figure 2.1. A federation with two federates](image)

To help the federates interact with the RTI there is an HLA runtime interface specification which describes the services that the RTI provides to the federates. These services can be separated into seven classes [6]:

- **Federation management** – The federation management provides services for operating and creating federations.
- **Declaration management** – The declaration management provides services for managing data exchange within a federation, these services are based on federates defining what data they require and what data they can send in the federation.
- **Object management** – The object management provides services for creating, deleting, and identifying objects.
- **Ownership management** – The ownership management provides services for managing ownership of attributes during execution.
• Time Management – The time management provides services for synchronizing data exchange in runtime.

• Data distribution management – The data distribution management provides services for routing data between federates during execution.

• Support Services – The support services provides miscellaneous services which can be utilized by federates, for example RTI startup and shutdown.

2.2.1 HLA Object Models

The HLA Object Model describes elements that can be shared in a federation. There are three different types of object models in HLA: Simulation Object Model (SOM), Federation Object Model (FOM) and Management Object Model (MOM).

The SOM specifies which types of information a federate can provide to an HLA federation and which types of information a federate can receive from other federates in an HLA federation.

All federations require a FOM. The FOM specifies what information can be exchanged during a federation execution. Object classes, object class attributes, interaction classes, interaction parameters and contents of the MOM can be found in the FOM. The MOM contains support for control and monitoring of federations (IEEE, 2010).

HLA does not define what a FOM or SOM should contain but it requires that documentation of both a FOM and a SOM is done in a standardized way using an Object Model Template (OMT).

2.2.2 HLA Rules

HLA rules define the basic principles of HLA. The rules are divided into Federation rules and Federate rules (Dahmann and Morse, 1998). The Federation rules are listed below:

• All federations must have a FOM that follows the OMT.

• All object representation is done in the federates during runtime and only one federate can own an object or attribute at the same time.

• All information exchanged between federates has to be handled by the RTI utilizing the HLA interface specification.

Individual federates have to follow some additional rules, the Federate rules:

• The information in the SOM must be documented using the OMT by each federate.

• Federates must use the information in their SOM to transfer ownership of object attributes, import and export information, update attributes and use the RTI to manage local time.
2.2.3 Performance in HLA

Performance measurements in HLA can be done by analyzing different aspects such as processor load, frame rate, data reliability and speed of data exchanges. These aspects are explained further in [7]. This report focuses on two specific aspects relevant to the thesis. Frame rate in the graphic visualization and Roundtrip Time (RTT) in federations.

The frame rate becomes interesting to study as the purpose of the plugin is to enhance the visualization of a simulation. Should this visualization get a low frame rate, objects get choppy movement and are hard to follow, in the report by Ping[7] a frame rate of 10 frames per seconds is defined as the lowest acceptable value. A lower frame rate would not give an enhancement of the visualization, thus it would not be any point in using the plugin should the frame rate drop too low.

Measuring RTT between federates is important to study the speed of the data exchange in a federation [8]. This is of course a very important aspect to study as data exchanges happening too slow within a federation leads to communication problems and faulty representation of the federation.

2.3 Tools and Environment

The tools used for the thesis work were Pitch Developer Studio, Pitch pRTI and the tank simulation mentioned in the introduction of the report [1].

2.3.1 Pitch Developer Studio

Pitch Developer Studio is a code generation tool which can generate HLA middleware in C++ or Java from a given FOM. The generated code makes all necessary code and error handling available to create a simulation based on the FOM specified when generating the code.

2.3.2 Pitch pRTI

Pitch pRTI is a RTI developed by Pitch which has a graphical interface for monitoring active federates and federations connected to it. It displays necessary information about a federation such as the federation name, the FOM in the federation and the names of the federates in the federation as shown in figure 2.2.
2.3.2 The Tank Simulation

The tank simulation has two types of federates. The first type is the Tank federate. The tank federate creates and controls a tank which communicates using HLA and can move around and shoot in a two-dimensional world. The second type is the Damage server federate. The damage server federate keeps track of tanks being damaged and how much damage they have received. If the tank simulation is going to work properly, one damage server needs to be connected to the federation, however no more than one is needed. There is neither a minimum nor a maximum number of tanks allowed in the federation. However, to see experience all the functionality of the simulation, at least two tanks need to be connected to the federation.

During federation execution, the tanks can rotate, go forward, go backward and shoot. Each tank has a unique ID and a damage count which counts up by one if they are hit. Figure 2.3 illustrates the visualization of the tank simulation.
Figure 2.3. Two tanks facing each other in the tank simulation.
3. Design and Evaluation Method

This chapter describes the design of the plugin and performance evaluation used to answer the research questions.

3.1 HLA-plugin in Unity

To answer the research questions, an HLA-plugin needs to be implemented in Unity. While implementing the plugin, alternatives for passing data between HLA and Unity needs to be explored to find out if using a plugin sets any limitations on what data can be exchanged and how. The tank simulation is the base of the plugin that is going to be implemented. The plugin shall enable Unity to visualize everything happening in this simulation in a 3D-world. To implement the plugin successfully, the tank simulation needs to be studied further, especially what types of data are passed within a tank federation and which data is relevant for Unity to receive to make a correct visualization. When implementing the plugin, the FOM that is used in the tank simulation is used in Pitch Developer Studio to generate HLA-code in C++ which can be the base of the plugin. The plugin code can then be written in a way where HLA-methods in the generated code can be utilized and exported in a DLL to Unity. This plugin handles the communication from HLA to Unity. The graph in Figure 3.1 illustrates the design and implementation process of the HLA-plugin in Unity.

![Diagram](image)

*Figure 3.1 The design and implementation process of the HLA-plugin*

In Unity, a script is written which imports the functions in the DLL. This script handles the communication from Unity to HLA. Since HLA is imported as a third-party library (thus a native plugin), implementing the plugin should follow the method described by Mike Geig in chapter 2.2.1.

The first step of implementation is to connect Unity to the federation using HLA. When Unity is connected, a suitable way to pass data between Unity and HLA should be found. If this is achieved, Unity can have knowledge of certain attributes of the tanks in the federation such as their position and
how much damage they have received. When this is known, Unity should be used to visualize these attributed using 3D-models placed in a 3D-world.

### 3.2 Performance Evaluation

After successfully implementing the plugin, its performance needs to be tested to find out if performance is lost when running HLA as a plugin in Unity. The performance is tested by measuring the two aspects described in chapter 2.2.3. To be able to interpret if the results are good or bad, the same tests are done in the tank application that is not run in Unity.

It is worth noting that these aspects can be affected by the machine running the simulations, especially CPU-load and frame rate. To ensure that the results are not misleading, all testing is done on the same computer with the following specifications:

- Processor: Intel® Xeon® CPU ES-1620 v2 @ 3,70 GHz
- 16 GB RAM
- GPU: Nvidia GeForce GT 630
- OS: Windows 10 Enterprise

Measuring frame rate in applications can be done easily using the tool Fraps. Malinga and Le Roux in [8] describes a method for measuring RTT by having a federation with two federates on two different machines and using a timer to see how long it takes for a message to travel between them. The setup they used is illustrated in Figure 3.2.

![Figure 3.2. A simple sketch of the testing setup](image)

In their setup, machine 1 sent an update which machine 2 receives and responds too. Machine 1 starts a timer when it sends the update and then stops it when it receives the response from machine 2.
Testing the RTT in the tank federation is done similarly to this setup. Timers are placed in the code and logged to a text file where the time taken for certain updates to be sent and received can be measured. The difference in the test setup and the one illustrated in Figure 3.2 is that all federates in the test setup are running on the same machine. This is because it is strictly communication speed within in the federation that is of interest in these tests, running the federates on different machines could allow for network issues not relevant to HLA to impact the test results.

3.2.1 Frame Rate Testing

Two test cases are examined for frame rate. In these tests, frame rate is evaluated but it is also of interest to verify that all data in the federation is exchanged correctly.

In the first test case a massive number of Tanks are spawned in order to see how Unity handles that. First 100 tanks are spawned, then 500 and lastly 1000. This is done twice, once by modifying a tank federate to create a lot of tanks and letting Unity discover these tanks using HLA. The second time, HLA is not involved, instead Unity simply spawns the tanks locally without being part of a federation. This is to be able to see if the test results are impacted by running HLA as a plugin in Unity.

The second test case is similar to the first. First 1000 tanks are spawned by modifying a tank federate. After spawning the tanks, updates are sent to them in different frequencies. First, updates are sent at 100Hz, then 500Hz and lastly 1000Hz. To be clear, one tank will receive one update at a time meaning that at 100 Hz the first 100 tanks will receive one update each during the first second. The camera in Unity is moved away from the area where the tanks are visualized because in this test case, it is not interesting to see how well Unity can render massive amounts of objects receiving updates. The interesting part is how the information in the updates are handled and how well the plugin runs while receiving massive amounts of updates to a lot of objects that are off screen.

3.2.2 RTT Testing

There are two types of tests used for RTT.

First, timers are placed at two certain places in the code. The first timer checks how long it takes to connect to the federation. The second timer checks how long it takes to register that a tank has fired a shot. This test is performed twice, once in the Unity plugin and once in a regular tank federate. The times measured in Unity can then be compared to the times measured in the tank federate.

The second test to verify that the HLA-plugin is not suffering from latency is by checking how long it takes for the plugin to visualize an update. In the test, a tank federate rotates its tank. The time when the rotation update is sent is logged. The visualization by the plugin is recorded and then the time of the frame where the rotation of the tank is visualized is logged as well. Subtracting the time the update is sent from the time of the visualization gives the time it took for the plugin to visualize the rotation. This is done twice to be able to compare the time it takes in two different federation executions and see if they are similar or not. If they are, they are reliable measurements.
4. Implementation and Test Results

This chapter presents the results of the implementation of the plugin and then the test results from the performance evaluation of the plugin.

4.1 HLA-plugin in Unity

When implementing the plugin, certain types of data were passed in different ways. Due to the extern “C” wrapper that was necessary for the plugin to work, certain data types could not be passed easily. Class objects could not be passed at all, the only way to access a class object was to use get and set functions for variables within the class. Struct objects needed certain workarounds to be passed. Simple data such as integers were easily passed between the Unity script and the C++ DLL, but strings needed a special workaround like the structs.

Structs were needed to pass data as efficiently as possible. One struct used to send data between Unity and HLA was the struct called tankinfo shown in Listing 4.1.

```csharp
struct tankinfo
{
    int x, y, orientation, damage;
};
```

*Listing 4.1. The struct holding all necessary information about a tank*

This struct exists in the DLL but it is necessary for the Unity script to read this data from the plugin to know where a tank is placed, which way it is oriented and how much damage it has received. For the Unity script to be able to do that, the struct needed to declared again in the Unity script and the struct object needed to be passed in a certain way which is depicted in Listing 4.2.

```csharp
[MarshalAs(UnmanagedType.Struct)] ref tankinfo
```

*Listing 4.2. Passing a struct object as a parameter in a plugin function*

Every time a struct was passed between Unity and HLA, it had to be done as is in Listing 4.2.

All simple datatypes that were not structs could be passed simply as return values of function, except for strings. This due to the C#-script and the C++ DLL handling strings differently. The solution to this was to pass a StringBuilder, which exists in C#, as a parameter from the C# script to all functions where a string needed to be passed between HLA and Unity. The StringBuilder was received as a char* in the DLL and then filled with all the characters that would as a total become the final string. Listing 4.3
shows the StringBuilder being sent from Unity to the function `get_last_joined` as an example of how this works.

```c

Listing 4.3 The function `get_last_joined` is called with a StringBuilder, its capacity and a tankinfo struct as parameters.
```

In the plugin where the function `get_last_joined` exists, the StringBuilder is handled as Listing 4.4 shows.

```c

Listing 4.4 `get_last_joined` takes a StringBuilder as a char*, its capacity as an int and the tankinfo struct as parameters.
```
The function, `get_lastJoined`, receives the `StringBuilder` as a `char*` in the C++ plugin file. The point of the function is to return the name of the tank that has most recently joined the federation to Unity. Inside the if statement, the `char*` called `lj`, which is the `StringBuilder` sent from Unity, is filled with the characters of the `string` called `temp` which contains the name. Having done this, the `StringBuilder` has all the characters and can be used to read the name in Unity.

Being able to pass this data between HLA and Unity enabled the DLL to use asynchronous listeners generated by Pitch Developer Studio to register all data exchange in the federation. Whenever an attribute of a Tank (for example its x-position) was updated, the listeners added the tanks unique ID to a list holding all tanks with updated attributes. On every frame, the Unity script checked if that list was empty and if not, Unity fetched the latest registered attributes for all tanks in the list and updated the visualization of these tanks.

Free assets from the Unity Store were used to visualize tanks, terrain, explosions and sky in 3D. The same tank simulation visualized in Figure 2.3 was successfully visualized in three dimensions using the plugin which is shown in Figure 4.1.

![Figure 4.1. The tank simulation from Figure 6 visualized in 3D using the HLA-plugin in Unity, displayed from the point of view of Tank95.](image)

### 4.2 Performance Evaluation

The testing was split into two categories in chapter three, frame rate and RTT. The results from the frame rate tests are presented first and then the results from the RTT tests are presented.
4.2.1 Frame Rate
The first frame rate test was to see how well Unity performed when creating a large number of tanks, both over HLA and locally in Unity. The frame rate when looking at an empty world was 109 Frames Per Second (FPS). Table 1 shows the results from creating the tanks using HLA and Table 2 shows the results from creating the tanks locally in Unity. The Frame Rate is written in the unit frames per second.

Table 1. Frame rate after creating tanks using HLA.

<table>
<thead>
<tr>
<th># of Tanks</th>
<th>Frame Rate (FPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td>500</td>
<td>12</td>
</tr>
<tr>
<td>1000</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 2. Frame rate after creating tanks locally in Unity.

<table>
<thead>
<tr>
<th># of Tanks</th>
<th>Frame Rate (FPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>28</td>
</tr>
<tr>
<td>500</td>
<td>8</td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
</tr>
</tbody>
</table>

The second frame rate test measured the plugin performance in regard to frame rate when receiving massive amounts of updates, however none of the tanks or the movement was rendered when testing this. Due to the camera looking at nothing, the FPS before sending updates was 338. The tests were done using three update frequencies, 100 Hz, 500 Hz and 1000 Hz.

At 100 Hz, the frame rate dropped between 290-310 FPS and all updates were handled properly. At 500 Hz, the frame rate dropped to 259 and then the program stopped responding until all updates had been sent. When all updates had been sent, the program became responsive again and all updates had been received and handled properly. At 1000 Hz the same result as was achieved.

4.2.2 RTT
Two kinds of tests were done to evaluate the communication speed in the federation. The first test registered the time it took to connect to the HLA federation and the time it to register that a tank in the federation had fired a shot. The test was run on both the plugin and a regular tank federate. Table 3 shows the results.
The next test measured the time it took for the plugin to visualize an update after it was sent. The test was run twice. The first time, it took 0.00165 seconds to visualize the update. The second time, it took 0.037552 seconds.

<table>
<thead>
<tr>
<th>Event</th>
<th>Time for plugin (seconds)</th>
<th>Time for Tank Federate (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connecting to HLA Federation</td>
<td>0.9604281</td>
<td>0.0</td>
</tr>
<tr>
<td>Register shot fired</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
5. Discussion

5.1 HLA-plugin in Unity

A plugin was successfully implemented in Unity. However, workarounds had to be made to pass certain data between the plugin and Unity. Not all datatypes and types of objects in the HLA code could be passed in their current state but all relevant data was possible to get across in different ways. This shows promise for running HLA as a plugin for Unity as a functional plugin was in fact implemented. The limitations on what data can be passed should not be too big of an issue in most cases, a lot of it comes down to how creative the developer is when finding ways to communicate what is necessary.

5.2 Performance Evaluation

The frame rate test results are discussed first and then the RTT test results.

5.2.1 Frame Rate

Testing the frame rate of the plugin showed that it does not work perfectly. Looking at Table 1 for the first test, the FPS dropped to 40 at 100 tanks which is acceptable to look at. At 500 tanks, the FPS dropped to 12 which is not very good and at 1000 tanks the FPS dropped to 7 which is also a bad result. Looking at the limit of 10 FPS which was used by [7], 12 is very close and 7 is below. However, comparing these results to the results of doing the same thing locally, it is clear that HLA is not what brings down performance. Creating the same amount of tanks locally in Unity performed even worse. Table 2 it is shows that the FPS dropped to 28 already at 100 tanks, then 8 at 500 tanks and 4 at 1000 tanks. These values are all lower but close to the values in Table 1, running the same test again might generate values closer to each other. The values of Table 2 shows that the problem that lowers performance is not creating the tanks using HLA. The problem that lowers the frame rate significantly is Unity's ability to render a lot of tanks in the 3D-world.

The second frame rate test also showed some problems. This time, rendering was not the issue since no tanks were rendered. At an update rate of 100 Hz, it performed well with a small FPS drop from 338 to somewhere between 290 and 310 FPS. All updates were received and handled properly, everything worked as it was supposed to. The problems started at 500 Hz and reoccurred at 1000 Hz. This time, the frame rate dropped to 259 FPS and the program stopped responding while updates were being sent. However, all updates were still handled correctly. This could have occurred due to several reasons. One reason could be that the Unity script fetches updates in chunks, if these chunks grew too large it could be possible that it occurred the program to stop responding if it took a long time to get these chunks. Another reason could be that although the tanks were not rendered, all the tank objects were still created in the 3D world and moved at a very fast rate. Performing the update operations on all these tanks at a high speed could be what caused the program to freeze as well. Although this is not a great result, it should not be interpreted as a flaw that comes from running HLA as a plugin in Unity. Once again, the performance loss can be traced to Unity itself as all HLA updates were properly communicated and received in all cases.
5.2.2 RTT

The RTT tests showed a lot of promise for running HLA as a plugin in Unity. The first tests showed that almost no performance was lost when communicating. The only time the plugin performed slower than the tank federate was when connecting to the federate. This could be due to the fact that connecting to the federation can at times be a bit slower than other HLA functions. It is worth noting that the plugin was not even a second slower than the tank federate. Also, when connecting to the federation, it is not too important that it happens instantly. When connected to the federation, the plugin performed as well as the tank federate. They both registered a shot being fired instantly, making the timers stop at 0 seconds.

Measuring the time from sending an update to visualizing it showed great results as well. At its slowest, it took 0.037552 seconds to display a tank rotating after the update had been sent. This is not at all noticeable to the human eye and gives a very truthful and accurate visualization of the simulation.
6. Conclusions and Future work

The first research question was:

- How does running HLA as a Unity-plugin affect the possibilities of data exchange in a simulation?

After implementing a plugin, it became clear that the simulation was not affected. It was able to exchange exactly the same data as it did without the plugin. The limitations only restricted how to pass certain data between the HLA plugin and Unity, this did not however restrict what data was exchanged in the federation.

The second research question was:

- Is performance lost when HLA is used as a plugin for Unity?

The test results taken from the implemented plugin showed that performance was not lost by running HLA as a plugin for Unity. The only performance drawbacks found could be traced back to Unity itself. The tests showed that all HLA communication was done without losing performance, thus not affecting the federation.

For future work, it would be interesting to explore the possibilities of implementing an HLA-plugin for a more complex FOM like the Real-time Platform-lever Reference Federation Object Model or the (RPR FOM)[9]. In this FOM more complex data types need to be passed in the plugin. When creating a plugin for the RPR FOM it could also be interesting to control an HLA entity and send updates from Unity to the plugin instead of simply listening to updates in the federation as the implemented plugin did.

Another point of interest could be to connect Unity to a web based interface where the HLA communication is published and subscribed to. This would allow to move HLA applications such as the one used in this work to be used as applications on phones for instance.

It would also be interesting to study if a generic plugin between Unity and HLA could be implemented where the plugin could be adaptable in regard to what FOM it is compatible with and what actions it can perform in a federation.
struct tankinfo
{
    int x, y, orientation, damage;
};

struct firecoordinates
{
    int x, y;
};

HlaWorldPtr _hlaWorld;
wstring ccrchost;
int crccport;

map<wstring, tankinfo>::active_tanks;
unordered_set<wstring>::tank_updates;
queue<wstring>::new_joins;
queue<wstring>::delete_queue;
queue<firecoordinates>::fc_queue;

bool fire_boolean = false;
bool tank_joined = false;
bool tank_deleted = false;
bool update_available = false;

boost::mutex map_lock;
class TankUpdateListener : public HlaTankManagerListener::Adapter
{
public:
  void orientationUpdated(HlaTankPtr tank, int orientation, bool validOrientation, int oldOrientation, HlaGetPositionPtr timestamp, HlaGetLogicalTimePtr logicalTime)
  {
    
    
    }
  void damageUpdate(HlaTankPtr tank, int damag, bool validOldDamage, int oldDamag, HlaGetPositionPtr timestamp, HlaGetLogicalTimePtr logicalTime)
  {
    
    
    }
  void updated(HlaTankPtr tank, int x, bool validOldX, int oldX, HlaGetPositionPtr timestamp, HlaGetLogicalTimePtr logicalTime)
  {
    
    
    }
  void updated(HlaTankPtr tank, int y, bool validOldY, int oldY, HlaGetPositionPtr timestamp, HlaGetLogicalTimePtr logicalTime)
  {
    
    
    }
private:
  ofstream w_log;
};

class NewTankListener : public HlaTankManagerListener::Adapter
{
public:
  void hlaTankDiscovered(HlaTankPtr hlaTank, HlaGetPositionPtr hlaTimeStamp)
  {
    
    
    }
  void hlaTankDeleted(HlaTankPtr tank, HlaGetPositionPtr timeStamp, HlaGetLogicalTimePtr logicalTime)
  {
    
    
    }
private:
  ofstream w_log;
};
class FireListener : public HlaInteractionListener::Adapter
{
public:
  void fire(bool local, HlaFireParametersPtr parameters, HlaTimeStampPtr timeStamp, HlaLogicalTimePtr logicalTime)
  {
    map_lock.lock();
    fire_coordinates.firePoint = {parameters->getX(), parameters->getY()};
    fc_queue.push(firePoint);
    fire_boolean = true;
    map_lock.unlock();
  }
private:
  wostream w_leg;
};
DLLexport void hlaconnect()
{
    ofstream log;
    log.open("log.txt", ofstream::trunc);
    try
    {
        _hlaWorld = HlaWorld::Factory::create();
        HlaTankManagerListenerPtr _ll(new NewTankListener());
        HlaTankValueListenerPtr _l2(new TankUpdateListener());
        HlaInteractionListenerPtr _l3(new FireListener());
        _hlaWorld->getHlaTankManager()->addHlaTankManagerListener(_ll);
        _hlaWorld->getHlaTankManager()->addHlaTankDefaultInstanceValueListener(_l2);
        _hlaWorld->getHlaInteractionManager()->addHlaInteractionListener(_l3);

        log << "Connected, listeners active\n";
    }
    catch (HlaConnectException& e)
    {
        log << e.what();
    }
    catch (HlaInvalidLicenseException& e)
    {
        log << e.what();
    }
    catch (HlaFmException& e)
    {
        log << e.what();
    }
    catch (HlaRtiException& e)
    {
        log << e.what();
    }
    catch (HlaInvalidLogicalTimeException& e)
    {
        log << e.what();
    }
    catch (HlaInternalException& e)
    {
        log << e.what();
    }
    catch (HlaNotConnectedException& e)
    {
        log << e.what();
    }
    catch (HlaSaveInProgressException& e)
    {
        log << e.what();
    }
    catch (HlaRestoreInProgressException& e)
    {
        log << e.what();
    }
}
```c
Dllexport void hla_disconnect()
{
    mep_lock.lock();
    active_tanks.clear();
    map_lock.unlock();
    _hlaworld->disconnect();
}

Dllexport bool get_updated(char* lm, int length, tankinfo* ti)
{
    map_lock.lock();
    if (!tank_updates.empty())
    {
        ofstream log;
        log.open("log.txt", ofstream::app);
        if (tank_updates.begin() != tank_updates.end())
        {
            wstring lastMoved = *tank_updates.begin();
            stringstream temp(lastMoved.begin(), lastMoved.end());
            if (temp.length() <= length)
            {
                strcpy_s(lm, length, temp.c_str());
            }
            else
            {
                log << "Wrong length of tankname when reading updates" << endl;
            }
        }
        ti->x = t.x;
        ti->y = t.y;
        ti->orientation = t.orientation;
        ti->damage = t.damage;
        tank_updates.erase(tank_updates.begin());
        map_lock.unlock();
        log.close();
        return true;
    }
    log.close();
    map_lock.unlock();
    return false;
}
```
void get_tank_delete(boolean)
{
    if (tank_deleted == true)
    {
        tank_deleted = false;
        return true;
    }
    else
    {
        return false;
    }
}

void get_fire(boolean)
{
    if (fire_boolean == true)
    {
        fire_boolean = false;
        return true;
    }
    else
    {
        return false;
    }
}

void get_joined(boolean)
{
    if (tank_joined == true)
    {
        tank_joined = false;
        return true;
    }
    else
    {
        return false;
    }
}

void get_firecoordinates(firecoordinates* fire_co)
{
    map_lock.lock();
    fire_co->x = fc_queue.front().x;
    fire_co->y = fc_queue.front().y;
    fc_queue.pop();
    map_lock.unlock();
}
```c
/// DLEexport int get_last_joined(char* lj, int length, tankinfo* ti)
{  
    if (!new_joins.empty())
        {
            ofstream log;
            log.open("log.txt", ofstream::app);
            map_lock.lock();
            wstring last_joined = new_joins.front();
            string temp(last_joined.begin(), last_joined.end());
            tankinfo* t = active_tanks.find(last_joined)->second;
            if (temp.length() <= length)
                {
                    strcpy_s(lj, length, temp.c_str());
                }
            else
                {
                    log << "Wrong length of name of joined tank" << endl;
                }
            ti->x = t.x;
            ti->y = t.y;
            ti->orientation = t.orientation;
            ti->damage = t.damage;
            new_joins.pop();
            map_lock.unlock();
            log.close();
            return new_joins.size()+1;
        }
    return 0;
}

/// DLEexport void get_deleted_tank(char* del, int length)
{  
    ofstream log;
    log.open("log.txt", ofstream::app);
    map_lock.lock();
    wstring deleted_tank = delete_queue.front();
    string temp(deleted_tank.begin(), deleted_tank.end());
    if (temp.length() <= length)
        {
            strcpy_s(del, length, temp.c_str());
            delete_queue.pop();
        }
    else
        {
            log << "Wrong length of name of tank to be deleted" << endl;
        }
    map_lock.unlock();
    log.close();
}
Appendix B – Unity C# Scripts

```csharp
public class TankController : MonoBehaviour {

    // private data
    private Vector3 new_pos;
    private Quaternion new_rot;
    private int damage;
    private string tankName;
    public GameObject tankInfo;
    private TextMesh text;

    // private TextMesh tankInfo;

    // Use this for initialization
    void Start() {
        tankinfo = new GameObject(tankName + "TNT");
        tankinfo.transform.position = new Vector3(transform.position.x, 3, transform.position.z);
        TankInfo rend = tankinfo.AddComponent<MeshRenderer>();
        text = tankinfo.AddComponent<TextMesh>();
        text.font = Resources.Load<Font>("Tank/Prefabs/Capture it");
        rend.material = text.font.material;
        text.fontSize = 1000;
        text.characterSize = 0.03f;
        text.color = Color.red;
        text.anchor = TextAnchor.MiddleLeft;
        text.alignment = TextAlignment.Center;
    }

    // Update is called once per frame
    void Update() {
        transform.position = Vector3.Lerp(transform.position, new_pos, 5 * Time.deltaTime);
        transform.rotation = Quaternion.Lerp(transform.rotation, new_rot, 5 * Time.deltaTime);

        tankinfo.transform.localScale = new Vector3(transform.position.x, 18, transform.position.z);
        tankinfo.transform.rotation = Camera.main.transform.rotation;
        tankinfo.transform.position = new Vector3(transform.position.x, 3, transform.position.z);
        text.text = tankName + "-" + damage;
    }
}
```
```java
public void setDamage(int new_damage)
{
    damage = new_damage;
}

public void setTankName(String tankname)
{
    tankName = tankname;
}

public void newRot(Quaternion rotation)
{
    new_rot = rotation;
}

public void MoveToNewPos(Vector3 newPos)
{
    new_pos = newPos;
}
```
public class tankhandler : MonoBehaviour
{
    public struct firecoordinates
    {
        public int x, y;
    }
    public struct tankinfo
    {
        public int x, y, orientation, damage;
    }

    [DllImport("tankdll64")]
    public static extern void hlaconnect();

    [DllImport("tankdll64")]
    public static extern void hla_disconnect();

    [DllImport("tankdll64")]
    public static extern bool get_tank_delete_boolean();

    [DllImport("tankdll64")]
    public static extern bool get_fire_boolean();

    [DllImport("tankdll64")]
    public static extern bool get_joined_boolean();

    [DllImport("tankdll64")]
    public static extern int get_last_joined(StringBuilder l1, int length, [MarshalAs(UnmanagedType.Struct)] ref tankinfo ti); // use this for initialization

    [DllImport("tankdll64")]
    public static extern void get_deleted_tank(StringBuilder lm, int length);

    public Transform target;
    public GameObject explosion;
    public bool run = true;
    public Dictionary<string, GameObject> tanks;
    public GameObject newTank;
    public int tankSpeed = 3;
    bool firstJoin = true;
    private List<string> tankList;

    void Start()
    {
        tanks = new Dictionary<string, GameObject>();
        hlaconnect();
    }

    void OnApplicationQuit()
    {
        print("quit");
        foreach(KeyValuePair<string, GameObject> kp in tanks)
        {
            kp.Value.Destroy(kp.Key);
        }
        tanks.Clear();
        hla_disconnect();
    }
void Update()
{
    if (get_joined_boolean() == true)
    {
        int joins;
        do
        {
            tankinfo ti = new tankinfo();
            StringBuilder str = new StringBuilder(100);
            joins = get_last_joined(str, str.Capacity, ref ti);
            if (joins > 0)
            {
                string l1j = str.ToString();
                print("l1: " + l1j);
                Vector3 pos = new Vector3(ti.x, 0, -ti.y);
                Quaternion rot = Quaternion.Euler(0, ti.orientation, 0);
                GameObject nT = Instantiate(newTank, pos, rot);
                if (nT != null)
                {
                    nT.name = l1j;
                    tanks.Add(nT.name, nT);
                    nT.GetComponent<TankController>().setTankName(nT.name);
                    print("NewTank: " + nT + " has been instantiated and added to list of tanks");
                    if (firstjoin == true)
                    {
                        tanklist = new List<string>(tanks.Keys);
                        print("Camera will now follow this tank");
                        firstjoin = false;
                        camera_Init(l1j);
                    }
                    else
                    {
                        tanklist.Add(l1j);
                    }
                    else
                    {
                        print("Instantiation returned null");
                    }
                }
            }
        }
        while (joins != 0);
    }
}
while(run == true)
{
    tankinfo ti = new tankinfo();
    StringBuilder str = new StringBuilder(100);
    run = get_updated(str, str.Capacity, ref ti);

    if (run == false)
    {
        run = true;
        break;
    }
    string last_moved = str.ToString();

    GameObject currentTank = tanks[last_moved];
    Vector3 newPos = new Vector3(ti.x, 0, -ti.y);

    currentTank.GetComponent<TankController>().MoveToNewPos(newPos);
    currentTank.GetComponent<TankController>().newRot(Quaternion.Euler(0, ti.orientation, 0));
    currentTank.GetComponent<TankController>().setDamage(ti.damage);
}
if (get_fire_boolean() == true)
{
    firecoordinates fc = new firecoordinates();
    firecoordinates(ref fc);
    Vector3 exp_pos = new Vector3(fc.x, 5, -fc.y);
    Instantiate(explosion, exp_pos, explosion.transform.rotation);
}
if (get_tank_delete_boolean() == true)
{
    StringBuilder str = new StringBuilder(100);
    get_deleted(tanks, str, str.Capacity);
    string del = str.ToString();
    if (del == target.name)
    {
        target_deleted();
        GameObject.Destroy(tanks[del]);
        GameObject.Find(tanks[del].name + "EN");
        tanks.Remove(del);
    }
    else
    {
        GameObject.Destroy(tanks[del]);
        GameObject.Find(tanks[del].name + "EN");
        tanks.Remove(del);
    }
}
if (target)
{
    transform.rotation = Quaternion.Slerp(transform.rotation, target.rotation + Quaternion.AngleAxis(00, Vector3.up), Time.deltaTime * 5);
}
if(Input.GetKeyDown(KeyCode.A))
{
  if (tanklist.Count != 0)
  {
    int index = tanklist.IndexOf(target.name);
    int next = index + 1;
    if (next < tanklist.Count)
    {
      camera_init(tanklist[next]);
    }
    else
    {
      camera_init(tanklist[0]);
    }
  }
}

if (Input.GetKeyDown(KeyCode.D))
{
  if (tanklist.Count != 0)
  {
    int index = tanklist.IndexOf(target.name);
    int next = index - 1;
    if (next >= 0)
    {
      camera_init(tanklist[next]);
    }
    else
    {
      camera_init(tanklist[tanklist.Count - 1]);
    }
  }
}
```csharp
void target_deleted()
{
    int index = tanklist.IndexOf(target.name);
    int next = index + 1;
    if (tanklist.Count != 1)
    {
        if (next < tanklist.Count)
        {
            camera_init(tanklist[next]);
        }
        else
        {
            camera_init(tanklist[0]);
        }
        tanklist.Remove(target.name);
        if (tanklist.Count == 0)
        {
            firstjoin = true;
        }
    }

    void LateUpdate()
    {
        if (!target)
        {
            return;
        }
        else
        {
            float height = 6.0f;
            float distance = 10.0f;
            float heightDamp = 2.0f;
            float rotationDamp = 3.0f;
            float wantedHeight = target.position.y + height;
            float wantedRotAngle = transform.eulerAngles.y;
            float currentRotAngle = transform.eulerAngles.y;
            float currentHeight = transform.position.y;
            currentRotAngle = Mathf.LerpAngle(currentRotAngle, wantedRotAngle, rotationDamp * Time.deltaTime); // Damps rotation of y-axis
            currentHeight = Mathf.Lerp(currentHeight, wantedHeight, heightDamp * Time.deltaTime); // Damps height
            var currentRot = Quaternion.Euler(0, currentRotAngle, 0); // Convert angle to rot
            transform.position = target.position; // set camera pos and distance to target
            transform.position += currentRot * Vector3.forward * distance;
            transform.position = new Vector3(transform.position.x, currentHeight, transform.position.z); // set camera height
            transform.LookAt(target);
        }
    }

    void camera_init(string id)
    {
        target = tanklist[i].transform;
    }
```

References


