Achieving Interoperability Between Gaming Engines by Utilizing Open Simulation Standards

Garratt Weblin
Pitch Technologies UK
5 Upper Montagu Street, London W1H 2AG
garratt.weblin@pitchtechnologies.com

Rithviik Srinivasan
Department of Computer Science, College of Engineering, Design and Physical Sciences
Brunel University London, Uxbridge, UB8 3PH, United Kingdom
1807980@alumni.brunel.ac.uk

Fang Wang
Department of Computer Science, College of Engineering, Design and Physical Sciences
Brunel University London, Uxbridge, UB8 3PH, United Kingdom
Fang.Wang@brunel.ac.uk

Keywords:
Game engines, Interoperability, Distributed Simulation, Unreal Engine, Unity, High Level Architecture

ABSTRACT: *With the increasing availability of game engines and game engine content, it is becoming more and more cost effective for new simulators to use these game engines as starting building blocks. With decades of research and development already done in the gaming industry, existing game engines, like Unreal Engine and Unity, make use of powerful gaming technologies like physics, artificial intelligence (AI) and sound engines. Many of these game engines offer regular updates to meeting the changing demands and to embrace the latest technical advancements. Making use of these existing technologies, the cost and time to market of new simulation systems can be greatly reduced.*

In a previous paper, Unreal Engine was used to create an Unreal based simulator using High Level Architecture (HLA) and Real-Time Platform Reference Federation Object Model (RPR FOM). Using the Unreal Engine brought its own benefits and complications, related to both general simulation issues and lack of support in the Unreal Engine. This paper will go into more detail regarding experiences using Unreal Engine and Unity, the benefits and problems faced, and the findings from this academic project exploring innovative approaches to game engine interoperability for future solutions.

An academic project in cooperation with Brunel University London undertook a study to replicate the work done with Unreal Engine within Unity. Using open standards, a Unity plugin was created to connect a Unity world to a HLA federation. This allowed Unreal Engine and Unity to connect to a HLA federation using RPR FOM and to communicate and interact with each other. As part of this study a number of tests were carried out using the Unity plugin. The plugin is able to send and receive object updates to achieve interoperability, this is also verified with the use of a cloud RTI. Testing on the plugin showed that the geospatial data conversion is accurate and Unity's performance is not affected by use of the plugin. As with Unreal Engine, Unity suffers from general simulation challenges, like coordinate conversions, terrain, and 3D models, as well as these, some of the issues not only exist between existing simulators and new game engine based simulators, but between the games engines themselves.

Today, game engine applications stretch beyond the commercial entertainment industry. Industry and academia frequently build interoperable simulators that are used for research in many different fields including aerospace, defense, medical, environmental, and cyber security. Utilizing advancements already made in modern game engines a head start can be gained in physics, visuals, and AI modeling and simulation. This enables much faster development than traditionally has been possible. Open standards provides the ability to integrate multiple simulators utilizing different game engines.

1. Introduction

Simulations have been used by various industries for many years. They allow companies to train individuals in safety critical systems without a risk factor. They are also utilized for modelling and simulating various real-world scenarios which facilitate research and knowledge acquisition. Distributed simulations allow companies to use individual components that suit their own needs and simulation scenarios. They differ from their parallel counterparts, which execute a simulation using a closely coupled approach, by adopting a more loosely coupled based architecture. Companies can use simulations running on different platforms and different geographic locations with ease. This leads to reusability of components which in turn increase flexibility and reduce costs of conducting simulations.

In recent times, many sectors including defense and aviation have increasingly been using game engines to implement and visualize simulations. There have been investigations into the leveraging of commercial game engines for multi-domain image generation [1]. They explained that existing systems intended for organizational and specific individual simulation needs were dated due to upgrade costs being high and time consuming. It is often seen that simulation technology advancements depend on budgets available to build simulators and train people to use them [2]. On the other hand, many commercial game engines are free to develop with and are regularly updated to meet the changing demands and evolving nature of the technology industry with innovation and invention being the backbone. They have suitable visualization and graphical features and allow simulation developers to build within a low cost and easier learning curve. They have various advantages such as reducing the time for development and research, with provision of all the desired tools required and more [3]. Usage of game engines, while having the problem of compatibility between existing simulations, enhances the simulation industry by enabling the creation of simulations for any number and any kind of scenarios with relative ease with the latest technical advancements. Thus, the development of serious games and simulations using commercial game engines has become prevalent.

However, it has been pointed out that the usage of commercial engines for distributed simulations using open standards has been generally overlooked [4]. Since distributed simulations involve multiple simulations running on different platforms, it is required that game engines be compatible with existing simulations and interoperate. One way this is made possible is with the help of the High Level Architecture (HLA) infrastructure. Organizations like NATO require interoperability to enable the use of modelling and simulation across all nations, for joint and combined distributed simulation application across all domains [5]. In NATO's case they have a distributed simulation system spanning across various nations. Distributed training is essential to NATO for cost efficiency as they are a large organization. Distributed simulations can also be used to produce a more realistic system overall. An example of this is the research conducted on building a framework for existing Commercial Off-the-Shelf simulation packages to integrate well and interoperate in a distributed simulation environment [6]. If, for example, Unity is used for its AR features and Unreal engine has an exceptionally good physics engine, we can achieve the best of both worlds by interconnecting the two.

Defense simulations have become invaluable tools since the military were able to train efficiently because of which many lives of service personnel were saved in major conflicts [7]. The effective and safe operation of high-tech systems and platforms require well trained people. Without simulation in defense training, operating defense systems would be highly inefficient. As such, it is important to achieve interoperability. Ensuring interoperability between various platforms is key to proper functioning of the various simulation systems. Therefore, it is essential that there exists a standard of interoperability between game engines to enable compatibility between different simulations to reduce overall development cost and time. Using open standards such as the high-level architecture (HLA), the extent of interoperability between simulations of different game engines can be identified.

In a previous study [8], a plugin for Unreal engine was developed to connect to a HLA Runtime Infrastructure (RTI) which is an implementation of the HLA API standard, usually accompanied by a graphical user interface to manage a federation during execution. This paper has designed and implemented a plugin that connects Unity to the RTI using the Realtime Platform Reference Federation Object Module (RPR-FOM) [9] to achieve interoperability between the two engines (Unity and Unreal). The following section of this paper introduces the relevant work on High Level Architecture and open standards, gaming engines, existing approaches and Applications and tools. Section 3 presents the design and implementation of the Unity Plugin, evaluates the extent of interoperability achieved and measures the performance impact of the Unity plugin. The last section concludes the paper.

2. Related Work

2.1 High level architecture and open standards

High Level Architecture (HLA) is a standard for distributed simulation. It is a network of simulations, known as federates, connected to a common federation. A federation comprises of interacting federates, a federation object module (FOM) and a run-time infrastructure (RTI). Communication between federates occur in the form of publish and subscribe services, where federates subscribe to specific information they are interested in and only receive that information from publishing federates. HLA is a generic and domain independent standard for simulation interoperability (IEEE, 2010). Every simulation has its own pros and cons. HLA allows groups of simulators to interact and be part of large simulations they might not have been designed to support. HLA removes the element of creating simulations that cover a wider variety of objectives by creating a structural platform where reuse and interoperability is built into the core foundation. This eventually leads to reduction in the cost and time required to develop a new virtual environment for newer purposes. Every simulation is different and therefore HLA uses a flexible approach in its implementation to promote reuse and interoperation among simulations.

The main components that make up the architecture of HLA are follows [10]:

1. **Run Time Infrastructure (RTI)** - this is where a federation execution takes place. The RTI enables the creation of a federation and connection/disconnection of federates to this federation. The services are related to information exchange, synchronization, and federation management. Services offered are explained further in the upcoming section.
2. **Federates** - individual units of a federation execution. Federates are simulations that are attempting to exchange data and interoperate with other simulations.
3. **Federation Object Model (FOM)** - it is the foundation for interoperability between simulations. It provides information on the objects and its attributes, interactions. Without this, data exchange across an RTI cannot occur. It contains the definition of data in the form of object classes and interaction classes pertaining to a distributed simulation's requirements and applications. A FOM can be provided as modules for better separation of concern, and to better support development by different teams. There are standardized FOMs, often known as reference FOMs, that can be extended using project specific FOM modules.

The HLA Standard consists of three parts:

1. **IEEE Std 1516-2010 Framework and Rules** - ten rules that define the responsibilities of federates and a federation and how the interactions among federates must be handled.
2. **IEEE Std 1516.1-2010 Federate Interface Specification** - it defines the runtime working of an HLA interface. It specifies the services offered by the federation to the RTI and by the RTI to the federation. This specification helps facilitate the communication between federates and the federation.
3. **IEEE Std 1516.2-2010 Object Model Template Specification** - it is a documentation of what kind of object models are used in HLA. Object models define the basis against which communication between federates can occur. Object models help standardize datatypes produced from different simulations as each simulation varies in the requirements of data and the type of data being produced. Object models are characterized by objects, interactions, attributes, and parameters. The objects and attributes of a model is based on the relevant requirements of a simulation which are guided by its ultimate intended purpose. Essentially, the main aim for an object model specification is to enable interoperability and reusability between federates. The types of object models include: Simulation Object Model (SOM), Federation Object Model (FOM).

2.2 Gaming Engines

The current interest to use game engines in simulation is not the first time they have made an appearance. Game engines have been powering simulations for over 20 years, having started with Marine Doom in 1996 which was used to train the United States Marine Corps. This used a modified version of the popular computer game Doom II. Currently, VBS [11] is a widely used virtual and constructive simulation system that is widely used all over the world. VBS is derived from the popular commercial game ARMA and has until recently been using the ARMA engine to power the simulation system. BISim have recently released the latest version of VBS, called VBS 4, which offers high fidelity graphics, a customizable weather engine, and built in terrain streaming using their terrain server (VBS World Server). With the in-built interoperability of VBS, it is an out the box simulation solution that can help train users all over the world.

There is a growing interest to explore the possibilities of building simulation solutions based on a number of available game engines. Unity [12] and, increasingly, Unreal Engine [13] are popular choices to those in the defence training sector. These game engines have been established over a number of years and are frequently used to develop both smaller indie games, and large AAA titles that generate millions of dollars in the gaming industry. Most commercially accessible game engines are free to develop on, with some like Godot [14] being Open Source, but operate different licensing models for their use outside of development.

Both Unity and Unreal Engine provide a user friendly editor for developing within their engines. This allows user with limited programming knowledge to create projects and interact with the engine. Unreal Blueprints is especially powerful, using draggable nodes to simplify the development process, lowering the barriers on companies to create simulation solutions with the engine. As with the currently accepted modular architecture approach taken by large systems, these game engines offer a component based software development approach. Allowing developers to focus on different features of the engine, by splitting the development between teams with different skill sets.

2.2.1 Game Engines in simulation

Due to the increase in usage of these game engines, there has been a push by the game engines themselves to cater more for a simulation based approach. Unreal Engine have been investing heavily in simulation and working with suppliers of simulations to make the engine easier to use, as well as provide models and affects to be used in simulation.

The use of game engines provides a strong foundation for development of new simulation systems. With decades of development already researched and implemented by these game engines, companies can make use of these powerful gaming technologies. Rather than developing pre-existing systems, like physics, artificial intelligence (AI) and high fidelity visual systems, time and cost can be saved by utilizing game engines.

2.3 Existing Interoperability Approaches

There are some existing solutions for connecting Unity to an HLA RTI. There is also a solution to convert Unreal engine data to Unity but not in real time. There seems to be no apparent existing project or solution intended for achieving cross game engine interoperability via HLA or anything in general.

2.3.1 VR-LINK Module for Unity to connect to MAK RTI

MAK Technologies is another company that develops solutions for modelling and simulation software. MAK have their own implementation of an HLA RTI called MAK RTI. MAK RTI allows simulation software to integrate, simulate and visualise virtual worlds in an interconnected synthetic environment. They have their own proprietary implementation of a plugin for Unity which allows it to connect to their RTI and interoperate with distributed simulations based on existing simulation and modelling protocols.

2.3.2 Unreal engine to Unity scene conversion

There is also a plugin developed for Unreal engine that allows for converting and transporting a scene developed in Unreal to Unity. This was done by Ciprian Stanciu of relativeGames [15]. However, this cannot be used for exchanging data entities in real time from Unreal to Unity. This is not an interoperability solution that can be executed in real time. It only allows the non-real time conversion of specific Unreal datatypes like character models to Unity.

2.3.3 HLA Plugin for Unity

Previous work was undertaken to design and implement a tank simulation using Unity and HLA [16]. However, the tank simulation did not use the RPR FOM and therefore did not incorporate complex datatypes for entities. In addition, interoperability between two engines was not explored at all and it was solely intended for developing and evaluating a plugin for Unity for a basic simulation using HLA. This project in a sense is an augmentation of the previous work as not only is a plugin developed for Unity for the more complex RPR FOM but also implementation of a way to publish entities from Unity to the HLA is also accomplished. This previous work only focused on listening to HLA and receiving entities.

The work presented in this paper, in addition to listening and discovering, also investigated and performed publishing of entities from Unity and receiving them in Unreal engine. Furthermore, a cloud based RTI was also used in our work for comparing the listening and publishing process using this plugin and evaluating the results.

2.3.4 GRILL DIS
The Gaming Research Integration for Learning Laboratory [17] are a research group that focus on the use of gaming education and learning. They conduct research and development on the use of games and simulation. They have recently released a plugin for both Unity and Unreal Engine that allows the user to connect their game to a simulation using the Distributed Interaction Simulation (DIS) standard [18]. While the number of support DIS PDU packets is currently limited, work is being done to increase this.

3. Interoperable Unity Plugin

3.1 Design

The structural overview of the Unity plugin and its connection to Pitch is shown in Figure 1. HLA API is generated in C++ using Pitch Developer Studio based on the RPR FOM. Custom code is written in C++ to handle API calls to HLA RTI and manipulate the HLA generated code in such a way to set it for export as a DLL. The code is then exported as a DLL to Unity and added as a native plugin. Code is written in C# to manipulate the DLL code in accordance with Unity's visualization and plugin logic. On calling the connect function from Unity, a federation is created. All utility tools required are also connected to the same federation. Data for entities is supplied by Pitch Actors Evolved. Unreal Engine is connected along with a Google Earth plugin. Entities being published from Actors will be received by Unity and their visualization would be updated by appropriate conversion of datatypes being received to a form suitable to Unity. The spatial conversion's accurateness is verified by comparing Unity's visualization with Google Earth. Google Earth displays the exact location and attributes of each entity. The visualization in Unity should match Google Earths in terms of the relative positioning of the entities. Interoperability is then verified by publishing entities from Unity or Unreal engine and receiving its data on the other engine.

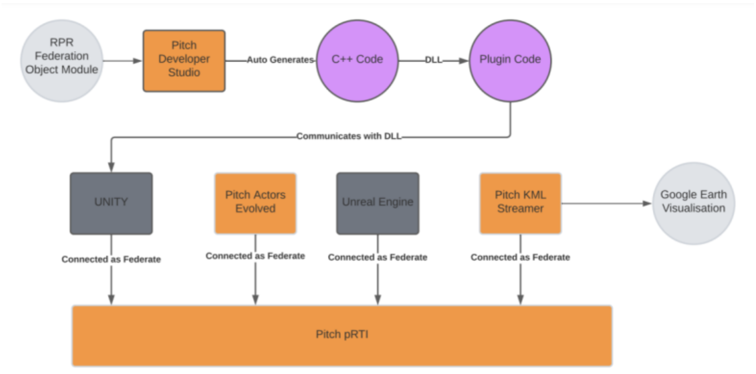


Figure 1. Overall Architecture of Interoperability Between Unity and Unreal

There are two types of plugins in Unity to import code written outside Unity: managed plugins and native plugins. Managed plugins consist of managed .NET assemblies that only contains managed code (Microsoft, 2021) written in high level languages like C# that can run on top off .NET. The execution of such managed code is handled by an application virtual machine known as CLR (Common Language Runtime). Unity's CLR is based on mono [19], a cross platform open-source subset of .NET framework. Mono allows Unity to support other platforms apart from Microsoft. The CLR takes care of memory management, security, garbage collection and more. As such a plugin can be considered as managed if it was written in C# and it uses libraries that are exclusively supported by .NET. Unity will not be able to access the source of a managed plugin due to compilation happening outside the boundary of Unity's execution. Since communicating with an RTI requires an HLA API that is written in C++ and is actually a form of middleware not associated with mono and traditional .NET assemblies, this paper uses native plugins instead of managed plugins.

Native plugins consist of unmanaged code or code that is written in languages like C, C++ and Objective C. Unity utilizes the features offered by the CLR that allows language interoperability through the System.Runtime.InteropServices library.

This essentially leads to integrating Unity with existing middleware or C/C++ code. Unity supports various plugin types including .dll, .cpp, .jar, .xlib and many more. In the work presented in this paper, code is written in C++ to handle HLA connections and receiving of entity data. The HLA plugin written in C++ is compiled as a dynamic link library (DLL) and added to Unity. DLLs are chosen because they can be loaded at runtime and dynamically linked or shared between HLA RTI and Unity. This is essential due to the concurrent nature of discovering entities and receiving attribute updates from RTI and updating the visualization for those entities in Unity. The HLA functions in the plugin will only be loaded into memory only when needed. This significantly reduces the overhead on Unity because of the HLA plugin and thus improves performance of any simulation. Additionally, DLLs promote reusability as the HLA code can be reused for other game engines or application as the code is modularized for HLA handling only. In accordance with Unity's handling of unmanaged code, functions are created based on individual processes that associate with unique HLA actions like connect, publish, get updates, and more. These functions are set up for export from DLL to Unity.

Visualization in Unity takes place in a process of constant communication between Unity and the DLL through the unmanaged code coupled as functions. For passing data between managed and unmanaged code, a process known as marshalling is required. Since C++ standard does not specify an abstract binary interface, interoperability between languages can only be achieved by using (extern "C") command to prevent name mangling issues.

In order to visualize entities, it is important to understand the data type of each entity's attributes. For accurate visualization, it is essential that Unity object attributes like transform.position and transform.rotation are matching the received data from HLA after conversion to a form suitable to Unity. For this the RPR FOM had to be studied. Each of the entities in the simulation has individual attributes that define the state it is in during simulation runtime. Attributes examples include damage status, marking, entity identifier, spatial, velocity and more. The spatial attribute is important for visualization. Its datatype consists of structs, the contents of which change depending on the type of simulation. Each spatial struct has the common attributes of World Location and Orientation. Orientation is based on Euler angles with axes convention being based on the coordinate system used.

The spatial attribute of entities contains a world location struct which contains XYZ coordinates based on the earth centered earth fixed (ECEF) Cartesian spatial reference system which is based on the spherical curvature of the earth. Unity's coordinate system is a left-handed Y up, Z North, X East Cartesian coordinate system based on a flat plane. ECEF coordinate triplet tends to be of large magnitude due to which entity positions are hard to compare and visualize in Unity due to the distances being relative to actual distances in earth. Unity uses a single precision floating points system for positions. ECEF's resolution can only be maintained by being in double precision. Direct usage of ECEF coordinates would therefore lead to loss in precision of coordinates when visualized in Unity and only make sense if a spherical object representing earth is used as a substitute for terrain. In that case, the earth object can be used to see various entities around the globe. However, this paper focuses on ground level visualization and thus it is necessary to manage unnecessary computational overhead by reducing the overall boundary of area being visualized while keeping the relative distances between entities intact. The flat plane in Unity can be approximated to a relatively flat patch on an earth's surface that is locally tangent to the ECEF coordinate location on a spherical context in a process known as planar or azimuthal projection. This creates a distortion free visualization. Conversion of ECEF geocentric coordinates to a local tangent plane coordinate system is essential for suitable visualization. The spatial attribute is converted using a utility function to a transformed spatial based on the North East Down (NED) local tangent reference frame. In NED, Z points down, X points north and Y points east. This requires flipping of coordinates when translated to Unity and applied as rotation for accurate reflection of orientation updates received from HLA. The transformed spatial consists of orientations with Euler angles and ECEF geocentric coordinates converted to WGS 84 (a version of ECEF that is used for GPS technologies) geodetic datum, consisting of latitude, longitude and altitude which will in turn be converted to Unity's xyz. This process of planar or azimuthal projection does raise the question of whether the positioning of entities would be accurate if a map streamer plugin were used for rendering real world location-based terrain in real time. A possible solution in that case would be to dynamically reduce the scale of the map and its terrain, which will be further investigated in the future work.

Figure 2 shows the design flowchart which guides the implementation process. The flowchart highlights the entire process's key steps ranging from connecting to a federation, discovering entities, and updating entity attributes. The design flowchart represents the coding pathway for both C++ and C#. A dictionary is maintained for all entities being discovered by the plugin. The dictionary consists of key value pairs of entity name and entity attributes. This is used to access a particular entity whenever an update for its attributes is received. Listeners are set up for discovering new entities or updating attributes. As and when updates are received, the dictionary is accessed, and the entity information is updated to reflect the new update received. Once Unity is connected as a federate, listeners check for new entities. Once discovered, entities are added to a dictionary and instantiated in Unity with the game object model being determined based on the entity class. Regular update calls between Unity and DLL occur to update the positions and orientation of each entity which will be converted to Unity's xyz using a utility script. Following an update, visualization will reflect the new positions and orientation of an entity.

3.2 Implementation

The overall communication between HLA and Unity is done in two parts. First between HLA RTI and C++ code and then between C++ code and C# code. The C++ and C# language interoperability are achieved through a process known as platform invocation or (p-invoke). P-invoke allows Unity to call unmanaged C++ functions running outside the control of Unity's runtime that are intended to communicate with an HLA RTI from DLLs. Representations of unmanaged functions are created in managed C# code. The DllImport() command imports C++ functions that were set for export. It takes in as parameters the DLL name and the entry point which basically points to the function's name in C++. The Interop boundary would take care of converting C++ datatypes to a suitable form for C#, which is marshalling. Unity uses the System.Runtime.InteropServices library for marshalling. The function representation in C# consists of parameters with the C# equivalent of the C++ parameter datatypes. Unity is the function call site where parameters will be marshalled to an unmanaged equivalent and placed on the runtime stack. The unmanaged function will then be invoked. As mentioned earlier, HLA API auto generated from developer studio is manipulated to create functions targeting specific goals of HLA actions and setting them up for export. Code written in C++ is passed on to Unity's C# through memory.

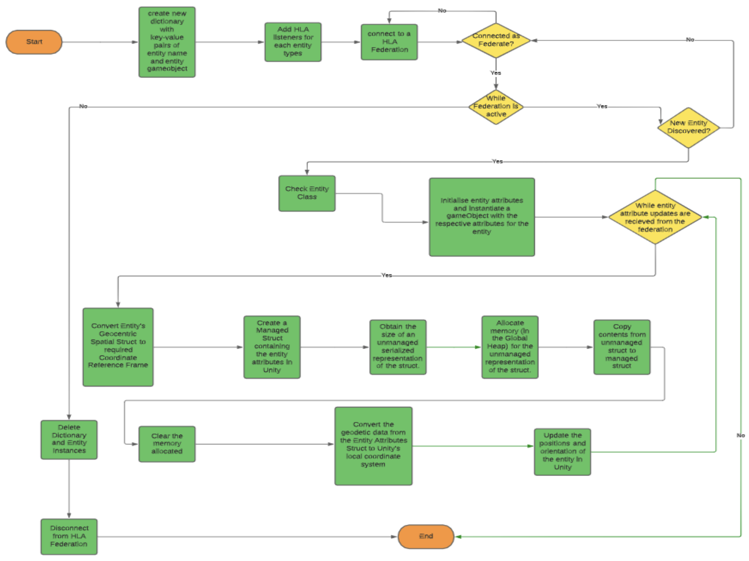

Start
create new dictionary with key-value pairs of entity name and entity gameobject
Add HLA listeners for each entity types
connect to a HLA Federation
No
Connected as Federate?
No
Yes
While Federation is active
No
Yes
New Entity Discovered?
Yes
Check Entity Class
Initialise entity attributes and instantiate a gameObject with the respective attributes for the entity
While entity attribute updates are recieved from the federation
Yes
Convert Entity's Geocentric Spatial Struct to required Coordinate Reference Frame
Create a Managed Struct containing the entity attributes in Unity
Obtain the size of an unmanaged serialized representation of the struct.
Allocate memory (In the Global Heap) for the unmanaged representation of the struct.
Copy contents from unmanaged struct to managed struct
Delete Dictionary and Entity Instances
Clear the memory allocated
Convert the geodetic data from the Entity Attributes Struct to Unity's local coordinate system
Update the positions and orientation of the entity in Unity
No
Disconnect from HLA Federation
End

Figure 2. Design Flowchart

Structs (Structures) were used as an efficient data structure to hold the entity attributes. A structure variable contains the data of the structure unlike classes that contain a reference to the data. Also, structures use stack allocation which is much faster than heap allocation used by classes. Stack allows the programmer to control the memory allocation and de-allocation which is suitable for this project. Entity updates and discoveries are tracked using listeners for each entity type. Listeners are created and activated on successful connection to a federation. As and when an entity is discovered from the HLA, it is attributes are initialised, and it is added to a dictionary mapping the name of the entity with the structs containing its attributes. This dictionary is used to access the required entity and its attributes easily. As and when an update is received by a listener, a queue containing the instance name of the entity being updated is updated. Thus, the latest entity to receive an update is taken and its attribute value is updated. The structure data includes datatypes that have common representation in both managed and unmanaged memory and are known as blittable and therefore do not require marshalling. However, the structure as a whole must be marshalled. This is done by having a copy of this structure in C#. Unity can receive the data required for updating the visualization by sending an instance of the struct in C# as a pointer to C++. Size of an unmanaged representation of the entity info struct is obtained and memory in the global heap is allocated for that size. Marshalling is done to convert the structure to a pointer and this pointer is used as a parameter for the unmanaged function getEntityInfoPointer being invoked from C#. As and when entities are updated, the latest entity to receive the update is accessed using the dictionary and its attributes are copied to the C# struct pointer. This modified struct is then passed back to C# using the Marshal.PtrToStructure command and the memory allocated is cleared.

Once the entity update is received from HLA, C++ code is written to transform the spatial attribute from ECEF system to the NED local tangent plane system. The struct variables are assigned to their corresponding lat, lon, alt and pitch, roll and heading variables from the update. Container scripts exist in Unity for each entity type. They hold the corresponding entity values like position and rotation. A publicly available utility script that converts GPS coordinates to Unity xyz was used for the conversion. The script uses a real-world origin point (0, 0) located near the Gulf of Guinea Africa as a reference point and returns a vector 3 containing the positions in xyz in floating point precision. The WGS-84 coordinates and orientation Euler angles are converted to degrees and saved in a container script in Unity. The world positions in Unity are obtained by using the GPSEncoder.GPSToUCS method. Orientations received from HLA are flipped according to Unity's coordinate system. The entity's, (in this example an aircraft), pitch, yaw, and roll axes conventions are also taken into account before flipping. This is because of the way in which xyz axes are defined in different coordinate systems.

Publishing entities from Unreal engine was a straightforward process. It could be done with the help of Pitch's Unreal engine connector. However, publishing an entity from Unity was not as straightforward and some steps had to be taken for it to work. Similar to how entity updates were received and sent in memory from C++ to C#, a DLL function was created to handle publishing. A typical publishing scenario involves a user submitting an entity's instance name along with its attributes. Since there are multiple entity types, it was essential to use if statements to check the entity class before going to the next step. There exists an HLA API function that takes in as a parameter the name of the instance to be used for publishing and creates a local instance with that name. The input is gathered in a struct containing the attribute values. This then passed in memory from C# to C++ and note that the struct is only passed in one direction from C# to C++ in this case. If an entity with the same instance name already exists, then that entity is retrieved for updating its attributes. Otherwise, a new entity instance with the name specified is created. A list is used to track published entities. Each entity has an updater class that can be used to update the attributes like marking, spatial, damage status and more. The final updater sends the received attributes from Unity to the HLA RTI.

A GUI was created so that simulations could be connected or disconnected to HLA RTI with ease. Specific functions were written in C# for facilitating the camera presentation during visualization. Two distinct types of camera views were created: 2D Top-Down Orthographic View and 3D Third Person Perspective View. This former was created to compare two entities and their visualization. The movement of the entities and their orientation in relation to the direction towards which they are headed can be seen as shown in Figure 3. The implementation of this view was done by getting the bounds of the camera to encapsulate the total number of entities in a list (in this case 2) and calculating the center point between them. Thus, the camera always moves to a midpoint between the two entities and visualizes both of their movement. The camera is bounded by calculating the greatest distance between the entities in view. This kind of camera display is useful for comparing entity visualization with that of Google Earth connected to the same RTI federation by using Pitch KML streamer.

The 3D third person view is useful when the number of entities is large. The camera sets its target as the first entity to be discovered. Entities discovered are added to a list. On pressing the A or D key, the camera can target the next or previous entity in the list. Figure 4 shows an aircraft as it steers left with this view. This view provides a grounded visualization and can be extended for applications involving realistic military simulation goals.

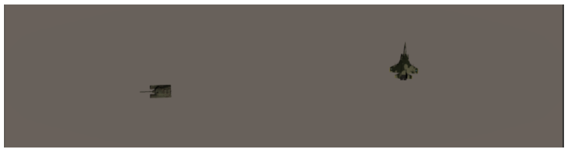


Figure 3. 2D Visualization in Unity

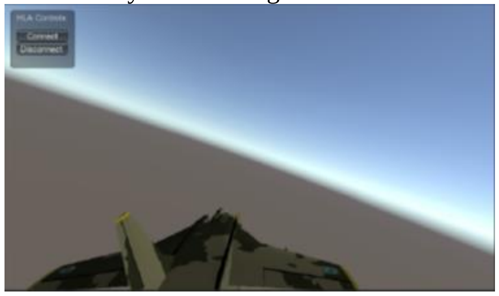


Figure 4. Third Person Shoulder view of entity

3.3 Evaluation

Performance evaluation was conducted by setting timers in different parts of the code to compare the amount of time taken by Unity to receive and update visualization from plugin against the time taken by plugin to receive and send data for different HLA activities. This was done in the form of a scalability test by increasing the number of entities for each entity type. With the same scalable idea in mind, a performance test was conducted to check the minimum frames per second considered as playable that Unity can produce with increasing number of entities with and without the plugin. This kind of testing allows to identify whether or not the plugin has an impact on Unity's visualization performance. The computer configuration on which this was tested are: AMD Ryzen 9 5900HS with Radeon Graphics 3.30 GHz; Nvidia GeForce RTX 3070 Laptop GPU and 16GB RAM.

For this design Unity plugin to be considered a success, it is vital that it does not negatively impact the visualization performance in Unity. Factors like frames per second, time taken for each update call and more help in the analysis process. Timers in different parts of the code provide the necessary metric to indicate which component takes up the most time in the visualization process. This is useful to know as it can be used to identify the major overhead inducing part of the plugin whether it is Unity's own rendering complexities or it is because of the HLA RTI's processes. Thus, the following approaches to evaluating the performance impact of the plugin were taken.

3.3.1 Time taken for different processes – Unity vs RTI

Timers are set up in both C# and C++. The C# timers indicate Unity's performance and time taken to reflect the update received from the plugin whereas C++ indicates the plugin's communication time with the RTI. Timers were set in such a way to provide data for the following processes: connecting time to HLA, discovery time for entities from RTI, discovery and spawning time of entities in Unity, one update call's time for RTI to send to Unity, time taken for Unity to receive the update from RTI and update the visualization, and the frames per second. The time taken for RTI is compared with Unity for the same to get a broad understanding of where each function stands in terms of performance overhead. Since Unity was connected to a federation for a definite number of seconds, it was necessary to calculate the average time for all these processes as the number of update calls are of larger magnitude. Each time, Unity was run for 10 seconds. The time taken was measured against increasing number of entities to identify what was the maximum count until which frames were deemed visualizable and this was done for each of the different entity types individually and collectively as well. The testing overview can be seen in Figure 5.

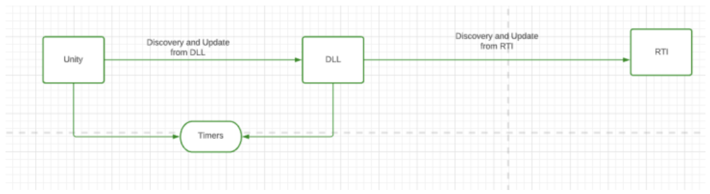
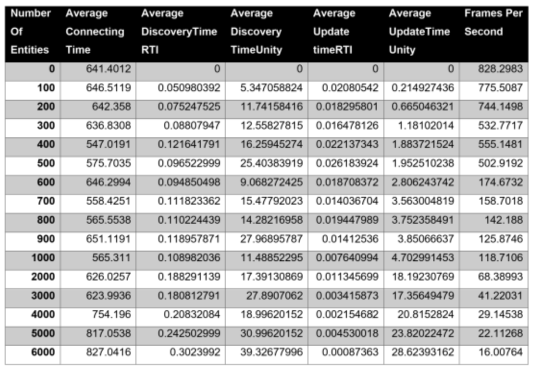


Figure 5. Diagrammatical Representation of Performance Evaluation

For each of the different magnitudes of entities under consideration, Unity was connected as a federate and was ran for 10 seconds before disconnecting from the RTI. The update rates from the RTI were 1Hz for all the different tests. Each test was conducted for 10 runs each. A run is defined by the one full cycle of discovery and update for all the total number of entities. Running the same processes multiple times indicate a huge difference in the values for each number of entities every time. Thus, the average of those values were calculated to obtain a suitable result for comparison. Figure 6 shows the various timings taken for the aircraft entity type and Figure 7 shows different timings comparison. All the data were recorded in milliseconds.



Number Of Entities	Average Connecting Time	Average DiscoveryTime RTI	Average Discovery TimeUnity	Average Update timeRTI	Average UpdateTime Unity	Frames Per Second
0	641.4012	0	0	0	0	828.2983
100	646.5119	0.050980392	5.347058824	0.02080542	0.214927436	775.5087
200	642.358	0.075247525	11.74158416	0.018295801	0.665046321	744.1498
300	636.8308	0.08807947	12.55827815	0.016478126	1.18102014	532.7717
400	547.0191	0.121641791	16.25945274	0.022137343	1.883721524	555.1481
500	575.7035	0.096522999	25.40383919	0.026183924	1.952510238	502.9192
600	646.2994	0.094850498	9.068272425	0.018708372	2.806243742	174.6732
700	558.4251	0.111823362	15.47792023	0.014036704	3.563004819	158.7018
800	565.5538	0.110224439	14.28216958	0.019447989	3.752358491	142.188
900	651.1191	0.118957871	27.96895787	0.01412536	3.85066637	125.8746
1000	565.311	0.108982036	11.48852295	0.007640994	4.702991453	118.7106
2000	626.0257	0.188291139	17.39130869	0.011345699	18.19230769	68.38993
3000	623.9936	0.180812791	27.8907062	0.003415873	17.35649479	41.22031
4000	754.196	0.20832084	18.99620152	0.002154682	20.8152824	29.14538
5000	817.0538	0.242502999	30.99620152	0.004530018	23.82022472	22.11268
6000	827.0416	0.3023992	39.32677996	0.00087363	28.62393162	16.00764

Figure 6. Various timings taken for the aircraft entity type

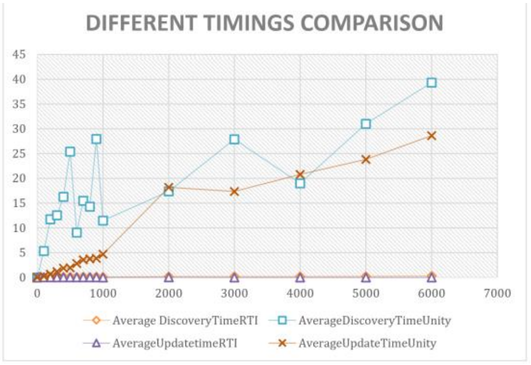
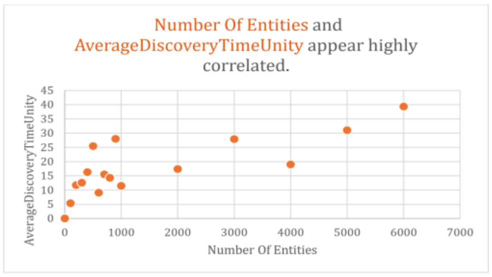


Figure 7. Timings Comparison

The average connecting time per run in Unity was significantly longer. There is a pattern of general increase in connecting time with increase in number of entities. This is consistent for all the different entity types. This is to be expected as a lot of processes including loading the DLL, finding the right port address for connecting to the RTI and various intrinsic processes occur. This does not negatively impact the visualization during runtime in any way. Average DiscoveryTimeRTI shown in figures 6 and 7 indicates the time taken for the C++ code to discover entities from a federation and send it to Unity. Average DiscoveryTimeUnity indicates the time taken by Unity to receive an entity from C++ and instantiate it in Unity's world. Based on the data collected, it can be seen that, the DiscoveryTimeRTI remains consistently around a small range of milli seconds from a minimum of 0.05 ms to a maximum of 0.30 ms. It generally tends to increase with increase in number of entities, but the difference is minimal. The UpdateTimeRTI is also similar. These values are significantly small and are not exactly visible to a human sense of perception. DiscoveryTimeUnity, on the other hand, is a little high with the lowest being 5.34 ms and the highest being 39.3 ms.



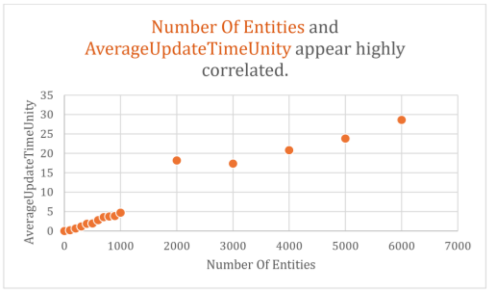
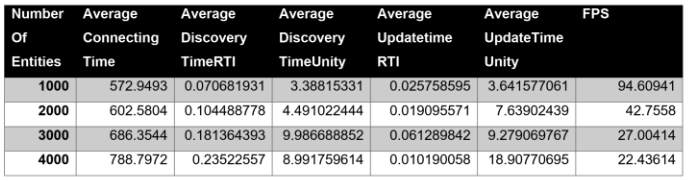


Figure 8. Correlation Graph between number of entities and Average DiscoveryTimeUnity (left) and between number of entities and Average Update Time Unity (right) -Aircraft

The number of entities seems to directly influence the discovery and spawning time for all entities in Unity as seen in Figures 8. This is also expected as an increase in number of entities increases the amount of calculations Unity has to perform in terms of rendering and graphical cost. Additionally, on comparing the averages of update time between RTI and Unity, similar results are obtained as the discovery part. The unity update time depends on the number of entities. Unity update time involves managing updates for all entities present in the world space and that is why there is some overhead in terms of performance. Each update frame, Unity has to process all of the entities present and update their visualization before going to the next frame. Based on the data, it can be seen that the DLL code communicating with the RTI has little to no impact and if any performance bottleneck is present it is because of Unity's own processing limit. This is further corroborated with the fact that the frames per second also decreases with increase in number of entities which is to be expected as update time of Unity per entity increases. At 6000 entities, the frames per second is the lowest at 16.00764. The discovery time and update time from the RTI remain consistent with the previous entity iterations in terms of range of value but the same cannot be said for Unity as the discovery and update timings have clearly increased with respect to the increase in entity count. Average Discovery and update times for RTI are largely uniform throughout whereas for Unity, there is a gradual but irregular increase with respect to number of entities. The jump between values for unity also seems erratic for different number of entities. This is possibly explained by the different processes happening within unity's ecosystem to process each object instance and render them in the camera view. Also, the timers are set up for timing loops that wait to discover an entity in unity and update its values. Because of this, the resources usage and

CPU time would vary as the loop iteration would depend on the availability of machine resources. Further data capture was conducted for other entity types as well which include ground vehicle, surface vessel and human. They showed similar results albeit differences in time values for Unity by virtue of the different 3D models being used. Further data collection was done using a combination of various entity types as shown in Figure 9. Each entity type contributed to 25% of the total number of entities. The Performance in Unity was impacted as the frames per second dipped below 30 fps for 3000 entity count itself. The Unity timings do tend to increase with increase in number of entities. However, the RTI times remain consistently small similar to the data in the previous tests. Thus, all this data indicate that the plugin or HLA has no negative effect on Unity's visualization.



Number Of Entities	Average Connecting Time	Average Discovery TimeRTI	Average Discovery TimeUnity	Average Updatetime RTI	Average UpdateTime Unity	FPS
1000	572.9493	0.070681931	3.38815331	0.025758595	3.641577061	94.60941
2000	602.5804	0.104488778	4.491022444	0.019095571	7.63902439	42.7558
3000	686.3544	0.181364393	9.986688852	0.061289842	9.279069767	27.00414
4000	788.7972	0.23522557	8.991759614	0.010190058	18.90770695	22.43614

Figure 9. Various timings taken for combined entity types

3.3.2 Frames Per Second with and without plugin

An additional test was conducted to capture the frames per second of visualization with a set number of entities in two scenarios: when Unity uses the plugin and gets entities from HLA, and when Unity is not using HLA and instantiates an equal number of entities locally. The results obtained were interesting. The disparity between the two scenarios as visualized in Figure 10 is important as it defines a new beginning for HLA plugins for Unity. Unity without the plugin performed poor compared to when the plugin was connected. In theory, a general assumption is made that a plugin that is communicating with an HLA RTI must have a lot of intrinsic processes that may increase overhead in performance. However, this data indicates that assumption to be wrong. It is almost as if the plugin boosted Unity's performance. This is also further justified by the evaluation conducted using timers. The plugin communication time with RTI was so small to be considered as negligible. Unity already has a lot of processes running in its background irrelevant to HLA.

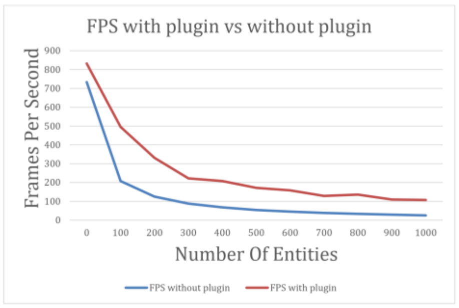


Figure 10. FPS Comparison with and without plugin

This could be a possible reason for the poor performance. Another possibility is that the overhead of the editor could be massive during play mode. But on checking a build, it showed similar poor performance without the plugin. A benchmarking was conducted to test for various scenarios to identify if C# or C++ was faster than the other [20]. The benchmark results indicated that mono tends to be slower compared to C++ or other .NET implementations for quite a few situations. However, after profiling both the scenarios, it was found out that Unity becomes GPU bound without the plugin as seen in Figure 11 indicated by Gfx.WaitForPresentOnGfxThread comprising 22.4% of the total processes. This does not exist when Unity is connected to the plugin as the CPU does more work while communicating with the plugin. When Unity spawns locally without a DLL, there seems to be an imbalance in the workload between GPU and CPU thus limiting performance. This is the more likely reason for the difference in frames per second with and without the plugin. When the plugin is loaded in Unity, a lot of CPU activities occur related to communicating with the plugin thus improving the performance of the visualization. No matter the case, it can be concluded that the plugin has no negative impact on performance.

Figure 11. Gfx.WaitForPresentOnGfxThread when plugin is not connected

4. Conclusions and Future Work

In this paper, a plugin was successfully implemented and loaded to Unity to enhance interoperability of simulations. Unity was able to connect to a federation and interact and exchange data with other federates connected to the same federation. Proper conversion of datatypes occurred in an efficient manner. A custom GUI for publishing was created to allow Unity to publish its own entity with attributes. Both Unreal engine and Unity were connected to the same federation. Entities published from Unity were received by Unreal engine and those published by Unreal were received in Unity. This was further verified by connecting Unity to a cloud RTI from Pitch where an entity published locally from Unity was received in the target computer's Unreal engine federate in the cloud. A detailed evaluation was conducted on the performance of the plugin. Based on the data captured, it was found out that the plugin showed no signs of having a detrimental effect on Unity's performance and thus ensured smooth visualization up to a certain threshold number of entities beyond which performance was deemed unsatisfactory. One caveat is that all the data were captured on the primary basis of not having a real time terrain map streamer for Unity's visualization. Due to this, Unity's performance might vary as the map streamer would be an additional overhead on Unity. Still, the data indicates that the HLA plugin would not be affected by this and would not affect Unity's visualization in any way.

The objective of achieving interoperability between the two game engines (Unity and Unreal) has been accomplished in this paper but it is limited in a preliminary sense. The publishing function developed consisted of only instantiating entities but not publishing and receiving interactions like weapon fire and munition detonation with entities from Unity to Unreal engine and vice versa. The work presented here does not include any sort of terrain data as a suitable map streamer that takes in real world map data and generates terrain based on that, at the time, the data was only available through a paywall. Manual creation of terrains is not necessary for this project as the focus is only on connecting to HLA and visualizing entities based on their spatial data. It would be interesting to see how terrain generation of real-world map data works in parallel with HLA entity communications. It would also be beneficial to use a map streaming plugin for Unity and validate the geodetic conversion on a real-world map-based terrain. Future considerations include exploring the publishing of interactions across game engines to fulfill the running of cross engine simulations that has minimal or no compatibility barriers. Another point of interest would be to explore the exploitation of this plugin to incorporate other modules of the RPR FOM [9] including underwater acoustics, logistics and more and focus on creating a serious game on the likes of ARMA but with the design principle being tied around cross game engine compatibility and interoperability by using Unreal engine's graphical fidelity and Unity's beginner friendly development environment. Additionally, a different FOM like the SISO Space Reference FOM [21] can be used to develop a unity plugin for integration and developing of space simulations.

Overall, gaming engines provide developers with a host of powerful tools for creating simulation systems of any size, reducing the cost and time to market for new simulators. Using open standards, like HLA, it is possible to connect different game engines and allow them to interact with each other. With the upcoming release of HLA 4, there are even more possibilities to develop simulators using game engine. Making use of features like federate protocol, game engines that are written in any language, like Pygame [22], can also be used. While game engines are powerful they still face similar challenges in the form of terrain correlation and scalability of distributed simulation. With new developments like terrain streaming and Unreal Engine's Mass Entity system [23], these could quickly become problems of the past.

5. References

[1] Dauble, J., Medford, A.L. and Frey, J.J. (2018) “Leveraging Commercial Game Engines for Multi-Domain Image Generation,” in MODSIM World 2018. Norfolk: ModSim World , pp. 5–11.

[2] Hiorns, B. (2021). Why defence simulation systems need interoperability. [online] Novatech Blog. Available at: https://www.novatech.co.uk/blog/simulation-interoperability

[3] Juang, J.R., Hung, W.H. and Kang, S.C. (2011) “Using game engines for physics-based simulations - A forklift,” Electronic Journal of Information Technology in Construction, 16.

[4] Ryan, M., Hill, D. and McGrath, D. (2005) “Simulation interoperability with a commercial game engine,” in European Simulation Interoperability Workshop 2005

[5] NATO M&S Master plan. Master Plan | NATO Simulation Standards.

[6] Park, J. (2005) A framework to model complex systems via distributed simulation: A case study of the virtual test bed simulation system using the high level architecture, ProQuest Dissertations and Theses. University of Central Florida.

[7] Wilkinson, S. (2019). Simulation in defence training: an essential tool. [online] Innovation News Network. Available at: https://www.innovationnewsnetwork.com/simulation-indefence-training/592/

[8] Suranga Wickramasekera, Boris Pothier, Garratt Weblin: “How Open Standards are making Game Engines more accessible for Serious Simulation applications”, September 2021

[9] Möller, B. et al. (2014) “RPR FOM 2.0: A federation object model for defense simulations,” in Fall Simulation Interoperability Workshop, 2014 Fall SIW.

[10] Wikipedia. (2021a). High Level Architecture. [online] Available at: https://en.wikipedia.org/wiki/High_Level_Architecture

[11] BISim VBS 4, https://vbs4.com/

[12] Unity Game Engine, https://unity.com/

[13] Unreal Engine, https://www.unrealengine.com/en-US/

[14] Godot, https://godotengine.org/

[15] Stanciu, C. Unreal to Unity Exporter. [online] Gumroad. Available at: https://relativegames.gumroad.com/l/unrealtounity

[16] Söderbäck, K. (2017). Design, Implementation, and Performance Evaluation of HLA in Unity.

[17] GRILL, Unity and Unreal Engine DIS Plugins, https://www.af-grill.com/open-source/opendis

[18] IEEE 1278.2TM-2015, IEEE Standard for Distributed Interactive Simulation - Communication Services and Profiles

[19] Mono Project, www.mono-project.com

[20] Qwertie (2011). Head-to-head benchmark: C++ vs .NET. [online] CodeProject. Available at: https://www.codeproject.com/Articles/212856/Head-to-head-benchmark-Csharp-vsNET

[21] Möller, B. et al. (2020) “SISO space reference FOM - Tools and testing,” in 2020 Simulation Innovation Workshop,

SIW 2020.

[22] Pygame, https://www.pygame.org/

[23] Unreal Engine Mass Entity, https://docs.unrealengine.com/5.0/en-US/overview-of-mass-entity-in-unreal-engine/

Author Biographies

GARRATT WEBLIN is a Software Developer at Pitch Technologies UK focusing on open standards based simulation interoperability for gaming technologies for defense, space and other applications. Garratt is a key contributor to Pitch Technologies recent developments on open standards based connectivity for HLA and Unreal game engine. Garratt studied Software Engineering for his BSc at Brunel University in the UK.

FANG WANG is a senior lecturer in the Department of Computer Science at Brunel University London. She received a PhD in artificial intelligence from the University of Edinburgh and worked as a senior researcher in the research centre of British Telecom (BT) Group. Dr. Wang's main research interests include nature-inspired computing, intelligent information processing, intelligent distributed computing and cognitive science.

RITHVIIK SRINIVASAN obtained a BSc (Honours) degree in Computer Science (Digital Media and Games) from Brunel University London in 2022. He is now studying for an MSc in at New York University, New York City, USA. He is specialized in digital games development.