Replicated Multiplayer in a First-Person Shooter

Jak Brierley

19 April 2017

A project report submitted in partial fulfilment for the degree of

Bachelor of Science in Computer Games Development

School of Physical Sciences and Computing

University of Central Lancashire

Abstract

Multiplayer games are becoming one of the top grossing entertainment platforms of the twenty first century. This report discusses using entity replication to transfer a multiplayer session’s entity data by using clients as hosts, without the need of a game server to decide and maintain the world state. The project that is attached to this report demonstrates entity replication via client to server by using high-level abstraction over network.

Attestation

I understand the nature of plagiarism, and I am aware of the University’s

policy on this.

I certify that this document reports original work by me during my University project.

Signature: Jak Michael Brierley **Date:** 19/04/17

Acknowledgements

I’d like to acknowledge a few important people that got me here. Laurent Noel & Gareth Bellaby for their outstanding contributions to teaching and advising me with projects these past few years.

Glenn Fiedler for making me fall in love with multiplayer games, and the guys at R* for believing in me.

Table of Contents

List of Figures

List of Tables

List of Listings

Introduction

1 Introduction

1.1 Background and Context

Multiplayer is becoming an important investment for many new releases. With micro transactions being a game's main business model, games such as Grand Theft Auto: Online can peak at $500 million dollars of revenue in a 12-month period (Ahmed, 2017).

To ship games with online functionality, the studio must provide dedicated servers to handle authoritative control of a games state. All clients must receive the current entity context frequently, and in open world games, the number of controlled objects can range to the thousands.

These state changes require high bandwidth and fast CPU processing time on servers purchased by the studio. This model is not cheap, and studios that make budget cuts may consider shutting down the servers causing a permanent closure of an entire game. A good example of this event happening is when GameSpy dedicated server's shutdown, killing all online services for up to 50 Electronic Art games (Phillips, 2014). This report investigates this problem, looking at client replication as an alternative technique of incorporating multiplayer worlds in one instance.

Client replication and older solutions such as RING are important techniques that reduce the maintenance cost of online services by simulating multi user virtual environments on multiple clients (Funkhouser, 1995). This in turn makes the demand for dedicated servers slimmer as servers do not depend on lots of resources for the same outcome, as clients are the underlying gameplay workers.

1.2 Scope and Objectives

This project is to investigate the overhead these systems depend on, and to compare them to similar multiplayer models that either benefit or disadvantage the longevity of software in regards to the expenses required.

To implement a robust approach, the project should describe object states across the network to form a multiplayer environment that gives clients replication access. The replication process should be authoritative server/client, where the server retains a true version of the game state while clients receive entity roles with read/write permissions (Epic Games, Inc, 2015).

The project should incorporate the previously discussed replication methodology, and should show the techniques used to build a client-replicated open world game. Techniques such as interpolation and dead reckoning should be a key focus within the project due to performance benefits (Smed, et al., 2002) and should demonstrate client side replicated entities interacting across the model.

The final goal for this project is to demonstrate that client replication can benefit studios by producing the same results as an authoritative client/server architecture would in regards to how gameplay co-exists with players.

1.3 Achievements

This project has challenged the problem of entity synchronisation within open world first person shooters.

Under the project implementation, the game network library (otherwise referred to as “netcode”) is a separate library that provides an interface for developers to use in various applications. Making the netcode standalone from gameplay.

The network library implements a standard authoritative server/client but extends it with the ability to define passive replication to connected users, with core utilities such as remote procedure calling for reliable function invocation.

Along with object replication, the projects framework also implements network snapshotting and delta compression to give fast, reliable network transmission for video games. With snapshotting being a sequential form of probing game states, the library does internal interpolation between vector and quaternion types.

The object persistence library has higher implementation that demonstrates networked objects, and replicated physics parameters. The players in the game request entities to construct on the server, and they then receive their controlled objects asynchronously on the next game snapshot. The player then applies local physics and sends this information back to the server for other players to simulate.

(Overview of Report, 1-2 lines per chapter)

1.4 Overview of Report

The second chapter after the introduction of this report covers previous literature about this topic of computer games. It evaluates historic techniques against present ones, and compares low-level protocols and the design of online games. The third chapter discusses design choices made within the project. A discussion takes place about why certain interfaces and object inheritance was chosen for maintaining game entities, and how the libraries are designed to be independent.

The fourth chapter discusses the implementation of the rendering engine used for this project, and how entities are stored in memory and rendered to the player. The fifth chapter discussed network control and how the high-level operations dispatch packets. A discussion also takes place about object persistence and how network objects are share their ownership. The sixth chapter elaborates on how object states and variables are serialised for networking. The implementation details about network variables and multithreaded access is also critically analysed.

The seventh section discusses the replication procedure of the network stack. This talks about ownership levels and how snapshotting is designed to be efficient. The eight section discusses testing on the project, and how performance matters.

Finally, the report is evaluated and future work is discussed in the ninth chapter.

2 Literature Review

2.1 Introduction

Integrating multiplayer is a challenge game developers must face when considering network playability. Single player games are isolated and rely on the client machine to evaluate decisions and to decide what moves or what events happen. This forward approach to handling gameplay completely changes upon integrating multiplayer, due to the attitude of game flow and its asynchronous control paths. The player's game becomes limited (i.e., a dumb client), and in most cases, cannot predict the future of the game state.

A review of deprecated techniques and historical examples of multiplayer will be analysed, as they served a purpose for building modern protocols used today. A discussion will critically evaluate the RING network architecture, as it was once of the earliest examples of distributed systems in game development.

In this section, a discussion about network tick rates will critically analyse requirements of coordinate interpolation in modern, fast pace games.

This section will also consider the usage of lockstep where information sent from clients to the server arrives in order and guarantees that network states will be internally consistent (Aldridge, 2011).

This review of existing literature discusses how multiplayer games maintain the same level of quality of a single player game. Many studios and independent developers (with successful titles) have concluded independent research into high-level networking, including entity replication. This chapter reviews current research made by these highly successful companies and independent sources, although this review does not cover security risks with cheating regarding these techniques.

2.2 History & Deprecated Techniques

2.2.1 Local Multiplayer Games

As early as 1958, local multiplayer games (same machine, multiple players) became a common solution to multi user interactivity. Game releases would divide screen space to give multiple perspectives, and different control schemes would control the assigned player entity. This technique was very easy to implement as it just used the single-system to reproduce a multi user experience (Madhav & Glazer, 2015).

Although local multiplayer was not a huge contributor to online gameplay, it did define the expectations for multi user gameplay. GoldenEye 007 (1997) split the game into four sections, allowing 4 local user co-op. This replayability and entertainment would define the next two decades of first person shooters.

2.2.2 Local Area Network Games

LAN gaming primarily is a topic spawned from the early implementations of multiplayer games on multiple machines as it was common to prefer it over WAN due to its constraints (Smed, et al., 2002) .

Varying from private LANs where people in a private home would play on the same network, to large LAN festivals where people pay to play with the other players as a social event (Ackermann, 2012).

Early multiplayer implementations found that the current state of the internet was lacklustre. Online games face three technological challenges, network bandwidth, network latency and host processing power (Smed, et al., 2002).

Research into multiplayer network traffic shows that analogue modems that were restricted to 56k would add an additional 120 to 160ms delay onto the round-trip time of packets (Färber, 2004). This delay can make fast action first person shooters impossible to play, and developers were stuck with this constraint.

Gaming over LAN defeated this constraint entirely. There was not any intermediary service between players; early as the 90s direct Ethernet gave 100mbp/s worth of bandwidth (NTT Communications, 2010), removing this dependency bottleneck from the equation.

2.2.3 Client/Server Protocol Design

Client-server architecture has become industry standard for most game titles, and most action games played online are modified client/server programs. This design orientates around a single authoritative server with “dumb” clients that deconstructs down to just processing input and rendering the current world state (Bernier, 2001).

Early games with competitive goals such as Quake (1996) and Counter Strike: Source (2004, using Source Engine) have been found to show that they both use a client/server protocol for multiplayer matches (Guzman, n.d. & Valve Corporation, 2005). The reasoning why this is preferred over the alternative peer-to-peer design is security flaws in it can give a cheater write access to the game state without an authoritative peer to validate the change (Baughman & Levine, 2001).

2.2.4 Lockstep

Aldridge (2011) describes lockstep as synchronising input or gameplay states so that simulations are in perfect deterministic sync. States such as input or current coordinate information can be synchronised so that their transition times are exact. It is common for RTS games to take this approach, as player input is synchronised and evaluated exactly at the same time. Lockstep is a peer-to-peer implementation, removing the need of a server between clients to supply and maintain game state.

The main disadvantage to this, which makes it unappealing for fast action games, is that lockstep in large player games will make each iteration wait for all players to synchronise their current input. If one player disconnects or responds later than everyone else, the time taken for the next iteration to happen is determined on the speed of player with the highest latency (treeform, 2016).

This approach also requires that network packets must be reliable and sequenced. Incoming changes from the current iteration must be processed for the current world state to be coordinated with other players. A dropped packet will make the current state one update behind other players, causing undefined behaviour. If a state update arrives in the wrong order, the world state will ignore a whole update causing a negative desync each time (Terrano & Bettner, 2001).

Enforcing reliability is not great for fast action games either. Coordinate data may be updated up to 40 times a second, and the overhead of protocols such as TCP (or reliable built on UDP) may cause detrimental network problems described further on this in this review.

2.2.5 RING Architecture

Before open world virtual environments became an industry standard, implementations of distributed entity replication came in the form of RING networking. The original purpose of this system was to support interaction between large amounts of users in a virtual environment with dense occlusion (Funkhouser, 1995). Entities with changes would only notify peers by calculating a zone’s potentially visible set, a form of occlusion.

Now with online games including competitive gameplay, latency can be a huge issue with disrupting a player’s experience. Network models with replicated entities or peer-to-peer transmission may do up to two jumps between hosts/clients for state updates before eventually arriving to a client’s game instance (Fiedler, 2014).

Funkhouser (1995) described that a disadvantage to RING was the applied latency of routing messages through a server. Because the model removes the direct peer-to-peer transmission, packets route through a server before arriving to recipients. This applies latency due to the fact of an extra destination and processing needed to retransmit to the target recipient.

2.3 Network Protocol Architecture

2.3.1 Transmission Control Protocol (TCP)

TCP is a reliable, connection orientated protocol that can simply application design with its reliability benefits. If a packet is lost, TCP will attempt to resend the contents to the destination until it arrives intact. It will not deliver previous packets until the lost packet arrives, making it sequential (Wu, n.d.).

RFC 761 was the formal definition for the TCP/IP protocol. The design goal of TCP/IP was to “build an interconnection of networks” and provide universal communication services over a diverse span of networks (i.e. wide area network). The purpose and main benefit of the protocol, which made it more attractive than UDP, was enabling long distance networks to share information reliably. UDP over large distance in its early stages was unreliable; packets could get lost in transit and even corrupt the contents. (Parziale, et al., 2006)

TCP also implements concurrent streams, supporting full duplex data streams to and from a host. Each connection has its own stream that can send data, meaning no extra overhead must exist to establish a second socket.

The biggest concept of TCP is its logical connection. Implementation of TCP uses a technique called sockets that open and close, where multiple data streams communicate to and from the target machine until a disconnection by either peer. This makes it connection-orientated as a general component of a system (Parziale, et al., 2006).

MMORPGs are one of the only sub-genres of games that employ TCP as one of the low-level network protocols (Suznjevic, et al., 2014).

2.3.2 User Datagram Protocol (UDP)

Formally defined in an RFC, UDP is “a procedure for application programs to send messages to other programs with a minimum of protocol mechanism” (Postel, 1980). This protocol introduced simplistic networked communication. It is the lightest and lowest latency protocol but does not ensure packets will arrive in a reliable state (Anand, et al., 2011).

The implementation concept of UDP is packet-orientated. At no single point is a connection established with UDP, packets send and in most cases, arrive at the destination. The protocol adds very little overhead and is the most efficient networking protocol (Wu, n.d.).

2.3.3 TCP or UDP for games

When investigating the technology that drive online gaming, it is important to investigate low-level protocols before attempting implementation. Without understanding popular protocol implications, high-level design will not naturally support gameplay as best as it could have.

TCP is a strong reliable protocol; it can ensure message delivery will happen and packet data cannot corrupt. This reliability benefit comes at a cost, and it can be serious for online games.

The overhead of TCP cannot work in time critical applications. TCP’s packet ordering and reliability checks schedule data sent after a lost packet into a queue. This queue is process only after the lost packet arrives, and the produced delay can cause serious online state game desync. Game networking mostly only cares about the current data, this backlog and time used to decode TCP packet frames makes this impossible. Multiplayer should be using both the benefits of TCP and UDP, but both causes’ issues as some data should be reliable and some should arrive fast.

When it comes to choosing the protocol, a hybrid between TCP and UDP needs to exist. Implementing TCP for reliable data and UDP for unreliable data seems to be an easy solution; however, these two protocols cannot coexist without causing issues. Studies into bandwidth interference found that TCP’s synchronisation stage negatively effects UDP reliability, suffering increased packet loss (Sawashima, et al., 1997).

Industry professionals recommend that game network protocols should be UDP. Important components from TCP need to exist as UDP wrappers, integrating key concepts such as reliability and sequencing into the same protocol (Fiedler, 2008).

2.3.4 RakNet

The RakNet library middleware is described on its repository as a “cross platform, open source, C++ networking engine for game programmers” (OculusVR, 2014). It acts as a UDP layer that acts as an interface for reliable packet transmission with datagrams, without the need of a TCP layer. The library can protect data, notifying the programmer if contents were changed mid-stream, whilst also establishing a connection layer like TCP to block unauthorised transmission (Jenkins Software LLC, n.d.).

This interface of network control, whilst also strictly only using UDP has minimal overhead. The layer provided by RakNet is designed to give the programmer exclusive control if they need it, but also not to overcomplicate a solution for game networking.

RakNet is a module-based library, with optional components offering further support for game networking. The core of the library is based on a server/client approach, but various plugins can support peer-to-peer networking with an arbiter pairing players to each other. This approach to extensibility makes this library very appealing to large studios who may have niche needs to networking.

Recent titles such as “For Honor” use deterministic input (a variant of lockstep) for their peer-to-peer combat (Yin-Poole, 2017). Even though this approach is highly controversial and out dated, RakNet already provides the peer-to-peer interface for interconnecting players. Deterministic input only needs to use the connection layer, without worry of packet reliability or sequencing issues.

Another huge benefit of RakNet is cross compatibility. The library’s codebase is universal and not specific to a single OS architecture. Platforms supported by RakNet cover a lot of the industries market share (newzoo, 2016), with console and mobile supported and tested as advertised on the project’s website (Jenkins Software LLC, n.d.).

2.3.5 ENet

ENet is a middleware network library, like RakNet. The libraries authors describe the project “to provide a relatively thin, simple and robust network communication layer on top of UDP” (Salzman, 2015). The library is designed to provide an event-driven model where client session management is made simple, providing a simplistic interface for multiple peer connectivity (Diehl, 2011).

The library’s simplistic approach does not make it too extensible as its competitor RakNet. Its interface serves as a simple peer I/O interface, where streams of data is sent using ENetPacket instances.

This event-driven approach, combined with the packet wrapper, makes ENet quite powerful for layering functionality on top. ENet handles reliability and sequencing via flagging packets, this means at run time certain information may separately dispatch differently. Even when large packets need sending, ENet handles fragmentation uniquely to the receiving peer. This flow control is powerful for optimising bandwidth, as the host should not expect clients to run the same hardware as MTU limits (maximum transmission unit of a packet) could vary a lot. (Diehl, 2011).

ENet has a uniform interface for both clients and servers, making peer-to-peer architecture possible. However, the support for peer-to-peer is not an official technique supported by the documentation. This undocumented functionality is something that makes this library unappealing for peer-to-peer games.

2.4 Online Game Architecture

2.4.1 Network Tick Rate

In a typical client/server network setup, the server simulates in discrete time steps called “ticks”. Formally known as a “tick rate”, it controls how many times a server will iterate over logic and network processing. Clients simulate at their own independent rate, otherwise known as the simulation rate. When clients receive incoming data, their game will process this information and attempt to render between frames (Valve Corporation, 2005).

This is the standard for most games with a client/server architecture model, as this technique guarantees that clients will receive a constant rate of data. This makes it easier to interpolate, as the next update is predictable just by knowing the server update rate.

2.4.2 Rendering Networked Gameplay

As previously descried, tick rate is the frequency a server will process logic and dispatch network messages. In a usual scenario, tick rate is usually between 30 ticks per second and 60. This rate is appropriate for network transmission, as bandwidth and server resources is scalar.

On the client side, the program flow of a game will do a step of logical processing and render to the screen. This requirement can make networking more difficult to implement as the network stack should be updated at a consistent rate (like the server tick rate), but frame rate is variable as not every hardware can maintain the same speed. Consequently, rendering must concurrently execute parallel to network stack updates, dividing network processing and rendering into separate sub-systems.

The rendering should be a standard graphics pipeline implementation, taking object matrices and drawing geometry based on the game state. The network sub-system will take events from the remote server, and update the shared game resources (entities, players) per received data. This is the simulation layer, and performs several tasks in relation to pre-entity rendering. The layer advances time, the rate of this should be a lot higher than the server tick rate, as a rate of 30 updates per second would reflect in the rendering sub-system as frame lag. The layer should also most importantly perform client side prediction (Frohnmayer & Gift, 1999).

The incoming rate of data received from a server will not meet the simulation rate. Most iterations in the network sub-system will be using the same data from the past three remote updates. The outcome would make moving objects in a world context choppy and jittery (Valve Corporation, 2005).

Standard game architecture introduces the need for interpolation, where data between a set of points reconstructs data that could have existed in the set. Because the network stack keeps hold of the previous update delta time, interpolation can take place for primitive geometric types. With this technique, positional and rotational data may apply it to provide smoother gameplay at the expense of calculating this each update.

2.5 Summary

The sources found have described the fundamentals for online network architecture. Most sources found support the same concept of a client/server model with periodic notification of state changes.

Sources found discuss implementation caveats, especially with low-level integration. Understanding each protocol (TCP & UDP) is important as both come with significant issues, and online gaming is incredibly dependant features found on both protocols. The research encourages developing TCP functionality on top of UDP, with supporting libraries ENet and RakNet using this for packet transit.

Material found about the two popular low-level libraries show implementation details, and identify usage of the software’s TCP/UCP blend. These libraries have avoided the problems of protocol usage in online gaming, and overall they are very appealing to this report’s project implementation.

Architecture research of online games has also found that update rates and the usage of sub-systems are an important concept for this project. The experience of how objects flow between coordinate systems in the implementation will be important for adding final polish to the game, and any physics should use these sources as a reference for networked movement.

3 Project Design

3.1 Introduction

This section outlines the architecture of the shared network library that drives object state replication, such as variable serialisation and underlying factory managers. The architecture of the rendering process is also covered and how the scene graph maintains the active world state.

This section also discusses interpolation and extrapolation algorithm design, and how it interfaces with networked entities and serialisation.

3.2 Network Architecture

3.2.1 Core Network Class

The act of transmitting an object's state from one host to another is known as replication. This technique depends on serialisation of object data, with information such as creation deletion and unique identification (Madhav & Glazer, 2015). These principles of networking object instances are heavily depended on in the project's network library, as the architecture used for object management uses standard factory classes parallel to network remote procedure calls.

As seen in figure 1, the implementation project will inherit an abstract data class depending on the needs of the application. Two classes are provided, NetSystemServer and NetSystemClient. These implement relevant transmission logic on top of the base NetSystem abstract class. The base system class also inherits an interface class with basic logic updates, such as transmission update calls and factory class lookup.

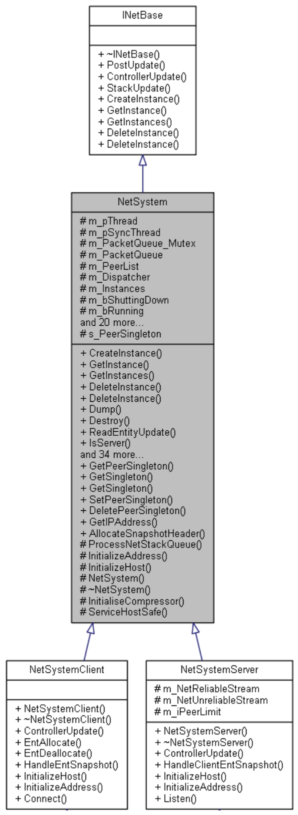


Figure 1 - Network System Class Diagram

This makes the network system encapsulated at the highest level, as most operations are done via factory managers defined in the system classes and the underlying class type will handle transmission and internal operations.

3.2.2 Virtual World

Within the networking library, data transmission is collected in “virtual worlds”. These are containers of entity managers, which contain entity compositions. This division of world states allows players to reference objects by their parent virtual world, for usage in culling to save bandwidth or to ignore entities from the render process. The implementation of the network library doesn’t use these exclusively, but defines a single virtual world with the ID “world” for storing entity managers for the game.

Virtual worlds aren’t type dependent as the contained entity factories are stored as their interface class. General interface usage such as entity building, or entity deletion doesn’t rely on type specifics. This interface usage is elaborated more in section 3.2.4 of this report.

3.2.3 Abstract Network Objects

Communication of class instances is important for world state integrity, as just having one instance missing or spawned as the wrong type will cause undefined behaviour and instability.

Game entities all derive from the base class NetObject. This base class implements a composition of a unique network ID for each instance and dispatches notifications for changed data attributes or object destruction. Network objects explicitly share ownership within the application, where the last reference will always be the containing factory manager.

Along with this, network objects can be inherited multiple times for project specific use. An example is within the implementation. CubeEntity is a derived class from NetObject, it contains shared attributes such as networked positions and rotations. From this as seen in figure 2, CubeClient and CubeServer both inherit from CubeEntity. The cube client processes render positions of objects, something that the server doesn’t require. Similarly, the CubeServer has its own initialisation routine, which sets the cube rotation on the server to its default position. This branching of networked objects is not a challenge for transmission as networked

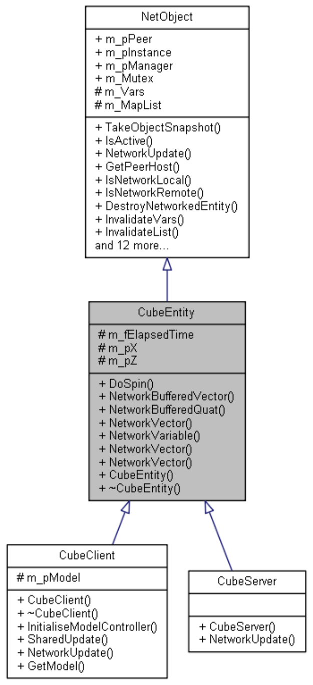


Figure 2 - Inheritance of a networked object

factory managers will know instance types and construct objects as the implementation defines.

3.2.4 Networked Factory Managers

The networked factory managers communicate class instance changes to relevant players. These managers help notify information about changes within the game's world, ensuring receivers are notified of changes. These managers are generic data types, and inherit IEntityManager interface to allow the network system to call update routines and network snapshot calls relative to the factory data type, which can be seen in figure 3.

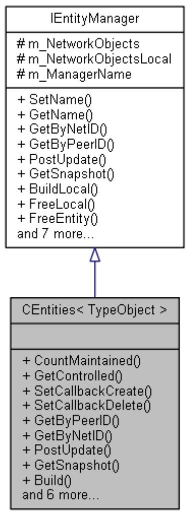


Figure 3 - Relationships of factory manager

Because this project has multiple codebases, there's been a lot of consideration into integrating consistent logic and event handlers across both the server and the client. This has meant having both codebases use a shared library that helps define common components and interfaces. The class RiNetBase is inherited by both RiNetServer and RiNetClient. Both deriving classes must provide an implementation for the factory managers. On initialisation of RiNetBase via the deriving class, the DefineManager function is called with all factory types used in the game. With this technique, the shared project can loosely implement managers for both the players and world cubes without needing to know the actual data type. This abstraction is powerful because the factory managers can be initialised by the deriving projects, but with consistent caller parameters across both projects by using the shared as the caller.

The only case where an object's last reference will be removed is when the usage count of an object is one (the manager), and the object is not marked as a server replicated object. Once the conditions for deletion are met, the manager drops its reference causing a destruction ensuring no part of the application has any dangling pointers. This design ensures objects are destroyed from within the factory manager, and that no dangling references exist.

All these changes are sent upwards from the factory to the network system to dispatch changes to players.

3.2.5 Network Variables

The implementation of networked cubes demonstrates the usage of network variables and their purpose. Common object variables are usually geometry information, such as position and rotation. The transmission of these data types can vary as non-POD types need to be serialised appropriately, and buffers such as strings need to be accessed and allocated differently to basic types.

The network library supports declaring standard class variables as “network ready” where other players listening for updates will receive changes regarding the network variable, and decide whether to use interpolation when accessing the data at runtime. By using macros, standard data types and structures can be wrapped with the NetworkVariable function, which will expand the incoming parameters to register with the network object.

As seen in listing 1, NetworkVariable declares an attribute of type “float”. When these network variables are initialised after the macro is expanded, when the parent network object is constructed the variable receives a reference to the parent which will then add the variable to a list for the instance to self-report its attributes at runtime.

```
class CubeEntity : public NetObject
{
public:
	NetworkBufferedVector(m_nPosition, false);
	NetworkBufferedQuat(m_nRotation, false);
	NetworkVector(m_nPositionSpawn, true);
	NetworkVariable(float, m_nMass, true);

	NetworkVector(m_nLinearVelocity, false);
	NetworkVector(m_nAngularVelocity, false);

	...
};
```

Listing 1 – [CubeEntity.h] Demonstration of network variable macros

This runtime abstraction allows any network object to know its own variable data at any point of execution within the program. Furthermore, because only the NetVar base class needs to be known for reporting network variable status, most non-complex data types can be generated as network variables using generic programming (see Appendix 1).

3.2.6 Data Serialisation

Because this project is multipurpose, the networking library needs to be able to handle various operations and distinguish between various incoming data types efficiently.

Reflection is the ability to maintain information about the internal state of data. Partially known or un-described data can be reflected to its original type as it was previously declared. Although C++ has very limited support for reflection out of the box, RTTI (run-type type identification) is partial reflection that depends on the application keeping track of class types of object instances (Roiser, 2003).

This approach is slow and would not have been suitable for this project as it depends on maintaining a list of object instance types in circulation, and for distributed systems this cause complications of maintain this list.

Instead the networking library builds each packet with an event type. The first two bytes represent the destination route to take on the receiver. This binds object serialisation and network updating into the same fundamental data stream, as new RPC calls can serialise with a unique route and object serialisation has reserved addresses to handle object changes (see figure 4).

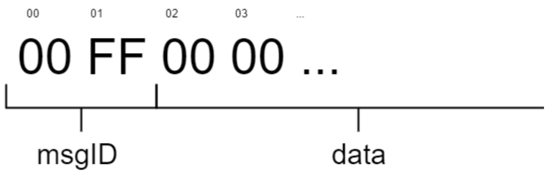
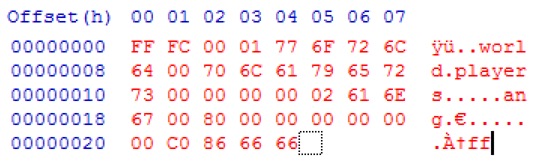


Figure 4 – An example packet of an event encoded with 00FF as its unique identifier

Object serialisation also follows this flow. Because "snapshotting" is based on sending changed data, there are reserved message IDs for creation, updating and deletion. When a packet is built to send updated data, the packet combines all updates into a single update. This "framing" extends the standard packet described above by firstly declaring in the data section how many frames exist and then packing



```
Offset(h) 00 01 02 03 04 05 06 07

00000000  FF FC 00 01 77 6F 72 6C  ÿü..worl
00000008  64 00 70 6C 61 79 65 72  d.player
00000010  73 00 00 00 00 02 61 6E  s.....an
00000018  67 00 80 00 00 00 00 00  g.€.....
00000020  00 C0 86 66 66           .À†ff
```

Figure 5 – A snapshot packet following the update serialisation structure

them after to form a full snapshot. See figure 5 and figure 6 as an example of a snapshot.

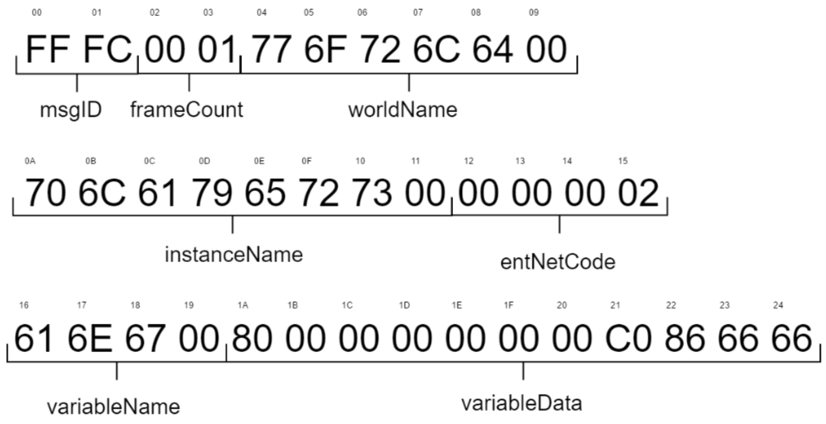

FF FC 00 01 77 6F 72 6C 64 00
msgID
frameCount
worldName
70 6C 61 79 65 72 73 00 00 00 00 02
instanceName
entNetCode
61 6E 67 00 80 00 00 00 00 00 00 C0 86 66 66
variableName
variableData

Figure 6 – Same data as figure 5, identifying data structure of packet

An alternative approach which is common in some networked systems is direct memory copying of structure data. This technique is very fast as no serialisation and packet building is necessarily needed, but the caveat to this is that memory alignment can significantly bloat or malform data.

Internal data within any application is not meant to be the smallest it can be or the cleanest to read. C++ itself has a feature called "alignment" where data type addressing and sequencing are expressed as the numeric address modulo a power of 2. CPU architecture has a lot of optimisation features that benefit from this positioning of data, integers stored on multiples of 4 can be understood and manipulated faster than an odd or misaligned location. This means that "padding" can happen between structure arguments (see listing 2 for an example). This padding of data is effectively undefined memory that is never recognised as accessible by a structure, but the overall size will cover it for allocating/freeing. Sending this data is considered unsafe and improper, padding contents aren't ever declared because of they're non-existant to the program.

```
class CPaddedStructure
{                               // Alignment Positions:
      char firstname_initial;  //   32-bit address: 0

      // (padding of 3 bytes to extend to 4, a good alignment figure)
      float bank_balance;      //                   4
```

```
     float bank_debt;          //                    8
     char  name[10];           //                    12

     // (padding of 2 bytes to extend to 24, a multiple of 4)
     int   age;                //                    24
};
// Binary representation where XX denotes a byte pad:
//    00 XX XX XX 00 00 00 00 00 00 00 00
//    00 00 00 00 00 00 00 00 00 00 XX XX 00 00 00 00
//
// Binary representation without alignment padding:
//    00 00 00 00 00 00 00 00 00 00 00 00
//    00 00 00 00 00 00 00 00 00 00
```

Listing 2 – C++ Example of a padded structure with 4-byte alignment

3.3 Game Architecture

3.3.1 Application Libraries as Modules

The scale of this project has required multiple implementations that eventually were compiled together. There are four of these modules: the network library, shared library, rendering library, client executable and server executable.

The network module is designed to handle all network transmission, giving abstract data types for an implementation program to declare and use. In this project, the shared library implements the network library with common functionality. Both the server and client inherit the shared library and implement project specific network functionality (server and client players, renderable nodes for client). The renderer is only included in the client executable, but provides an interface for rendering game entities and setting up the window, as seen in figure 7.

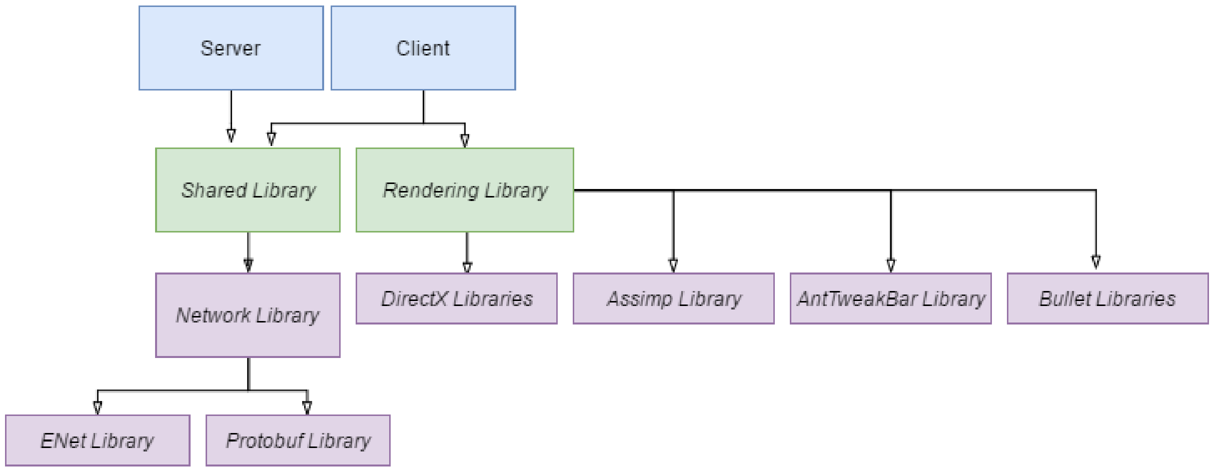


Figure 7 – A dependency overview of the total project

This splitting up of the rendering library is important as isolating its components and interfaces from the server will free up the global namespace and remove it as a dependency. If the rendering library is compiled, the server doesn't need to relink the library (or recompile) as it won't need to use any of its resources.

3.3.2 Scene Graph

A scene graph is a representation of the state of a game's world at any given time, usually represented as a tree like structure (Eberly, 2006). The renderer made for this project uses scene graphs to model the current world's state. Entity templates are always owned or referenced (using shared ownership) by scenes, and so are the associated entity references. With this, when a scene is swapped or loaded, the previous scene's entity compositions are lost.

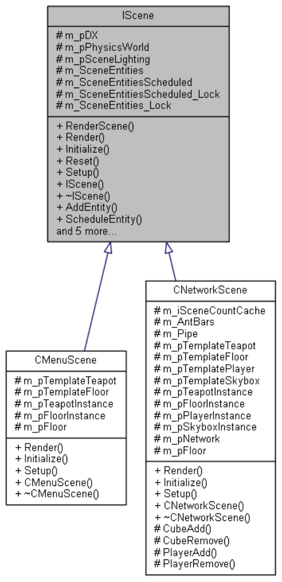


Figure 8 – Inheritance diagram of IScene

Within this game there is extensive usage of the scene graph. Because the game's network scene only exists after core game components are ready, the network library can attempt to connect and start receiving the world state. See figure 8 for the inheritance diagram of the IScene abstract class, showing that the network scene manages game specific components whereas the IScene class manages internal operations and entity ownership.

A scene is also responsible for updating entity logic and registering new instances to be sent to the renderer. When an object is instantiated, a call to the base IScene::AddEntity method must be called to register its ownership to be persistent throughout the scene's lifetime. This design is to allow the implementation to register objects at runtime, and for the abstract IScene class to manage object lifetime and usage within the renderer.

3.3.3 Asset Management & Scene Nodes

Resource allocating and re-usage is important for any time critical system, so the game engine made for this project uses an inheritance based templating system. As seen in figure 9, all assets are composed of BaseAsset. Base assets all maintain self-owned data streams which have unique ownership of the actual resource. Each time an asset is depended on, a call to BaseAsset::Build() makes a new cloned reference to the BaseAsset, making the data only clean up when the final dependant destroys. This shared model is essential for templating of entities, as a scene may depend on a ModelAsset but current has no instances active. The parent scene graph keeps a single template clone for instancing, so that when the scene is destroyed the BaseAsset is informed that there will be no further usage and to decide to free the memory or not.

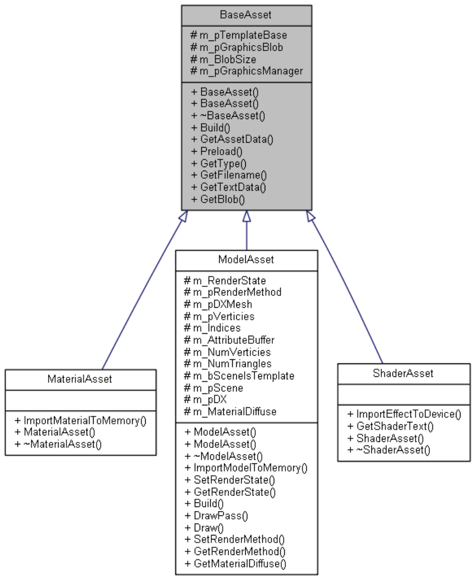


Figure 9 – BaseAsset inheritance diagram showing various types of game resource types

Instances of BaseAsset's are made using the RiGameNode abstract data type (see figure 10). This class wraps class usage with a reference to the model used. Then when it comes to the render process, the RiGameNode reports its underling ModelAsset instance which also reports its ModelAsset parent template.

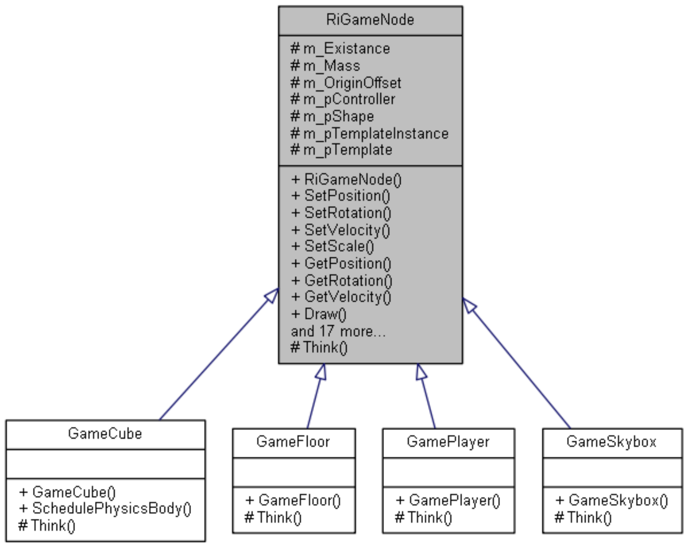


Figure 10 – Gameplay class types and their parent class

3.4 Algorithm Design

3.4.1 Interpolation

Data received by a client or server will be at a fixed rate. This rate is called the snapshot tick rate, and for the project it is set at 60 ticks a second. Because of this steady rate, the client will be limited to how many frames an object can be drawn in an updated state. For example, if the client is rendering at an FPS of 120, 60 ticks per second will show visible gaps between each snapshot. If the tickrate was lowered to 20, the rate of each entity coordinate update would be visibly 20 frames per second.

This problem can easily be solved by using interpolation between each snapshot. Network variables declared with “NetworkBufferedVector” (or the quaternion equivalent “NetworkBufferedQuat”) store extra details about the previous frames received. Because the last four position updates are stored, an interpolation algorithm is applied to produce clean, renderable position updates that look visibly smooth at a variable frame rate.

These vector updates use linear interpolation to create data points between each render call, storing the time passed on each invocation until the next snapshot arrives. Then it gets reset to restart the interpolation to continue transitioning from the current point to the next point.

$$\mathbf{from} + ((\mathbf{to} - \mathbf{from}) * \boldsymbol{time})$$

Figure 11 – Linear Interpolation

The equation of a linear interpolation with coordinates is taking the direction from the original starting point and scaling it between 0.0 and 1.0. This allows a new coordinate to be extracted from any point in time. Producing a new data set in real time creates a smoother experience, as the render positions of an entity can keep up to date with the current graphical frame time.

For quaternion interpolation, this project is using a geometric formula to produce a new data set of quaternion angles at a given time. It follows a similar implementation to linear interpolation with vectors, but considers that the resulting quaternion must be a curve within quaternion space. Deciding whether to use linear interpolation on its own for angles wasn't a difficult choice to make.

Gimbal lock is a situation where an angle represented in Euler loses an axis of rotation. This issues arises when a degree of freedom is lost in the general rotation matrix (Dam, et al., 1998). Rotation with quaternions may eventually fall under this situation if an angle is linearly interpolated through one of these situations, causing undefined results.

Because of the limitations with standard lerp, the algorithm chosen for networking rotation is spherical interpolation. This variant avoids issues related to Euler rotations by incrementing alongside an arc to the final quaternion value, see figure 12 (Shoemake, 1985).

$$Slerp(q_1, q_2, u) = \frac{\sin(1-u)\theta}{sin\theta} q_1 + \frac{\sin u\theta}{sin\theta} q_2$$

Figure 12 – Spherical interpolation between two points described as a 4D arc (Shoemake, 1985)

3.4.2 Extrapolation

Update rates within a networked application may vary due to a handful of issues relating to online connectivity. The connection between a player and the server may be briefly disrupted, delaying any scheduled snapshot buffers. This delay often leads to inconsistent outcome in games where data is real time and needed in quantity. Dead reckoning is an attempt at reducing visible effects of network induced delays, and this project uses multiple prediction schemes to reduce visible interference (Pantel & Wolf, 2002).

Buffered vectors used in the project not only just use interpolation algorithms to determine new data sets, but attempting to read the “live” value of an entity without receiving the next snapshot at that time will follow the same path until a max distance is reached. The same algorithm shown in figure 12 is used, but clamps it to a maximum distance of 1.2, which is the maximum extrapolation percentage. This adds an extra 20% as prediction for the next expected position update.

Because entities in the game are physics based, a Taylor series can represent and predict future entity positions. The library used for simulating rigid body dynamics is Bullet Physics, an open source project for integrating physics systems. Bullet takes parameters such as angular velocity, linear velocity and mass for real time physics simulations. In this project, these values are networked alongside positional data. Each player calculates the same physics outcome but only if the world state is mirrored and up to date. This creates an almost exact replica of the world across multiple machines even if network problems briefly arise.

3.5 Summary

The design of the project is aimed towards a modular and component based hierarchy. Networking is its own collection of object managers and instances, and it interacts with a similar system for gameplay. The interaction between the two systems is designed to be mutually exclusive.

Data transmission within the networking module also has a defined standard, and expects to follow this when encoding and decoding messages. Whilst the networking manager handles serialisation, its interface is meant to be simple enough that new class types can inherit and start replicating instances without any concerns about management of the object.

4 Rendering Engine

4.1 Introduction

The onscreen rendering engine made for this project was designed for a dynamic and abstract approach to real time game systems. This meant the requirements were primarily focused on having multithreaded access to game instances, whilst also having an important rendering sub-system to take internal states and to paint them to the user's window. The interface used between the application and the GPU is DirectX, a Microsoft based programmable pipeline. The link between these two systems together is the custom made rendering engine that was specifically made to meet the criteria requirements described above, whilst managing their resources appropriately.

4.2 Assets

For DirectX to draw geometric data, various amounts of assets are sent to the device to prepare each frame. Meshes are compositions of vertices and indices which build geometry; these compilations of data are stored in .X format and are parsed by a third-party library called Assimp.

Assimp is a multipurpose library that can convert most mesh formats to internal buffers that can be read by a DirectX device. Once this data is passed to the GPU, the current mesh's texture is also extracted using this library and applied to the current shader technique.

Shaders are also a type of asset in this project as they are represented as High Level Shading Language files (HLSL). These files are loaded at runtime by compiling the contents and sending it to the GPU.

Because assets are templates, their contents are instanced and referenced only once. Rendering a set of blocks in the game will only have one mesh and one texture per material associated with all instances.

4.3 Graphics Pipeline

Sending the game resources to the GPU depends on certain procedures to be declared by the program. These set of instructions are called the graphics pipeline, and are important for interpreting input data and producing various types of output. As seen in figure 13 as described by Microsoft, there are a lot of options and stages that can be implemented throughout the pipeline. This project only implements the vertex shader stage and pixel shader stage as other parts are only useful for topics such as tessellation, or parallel programming.

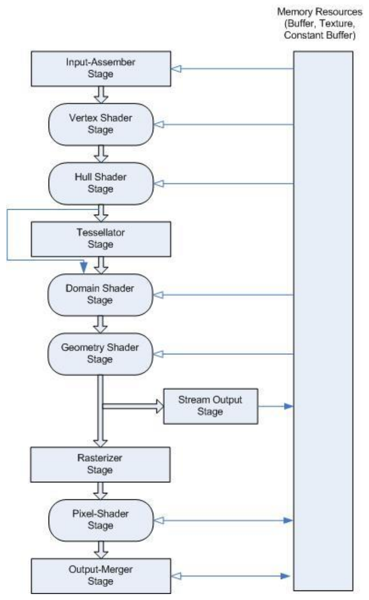


Figure 13 – Graphics Pipeline (Microsoft, n.d.)

The rendering pipeline does not need to be fully implemented, but certain stages are needed to support drawing textures and polygons. The stages that are required in this game's implementation are the shader techniques for both the vertex and pixel shader.

It was important that the vertex shader was implemented as core functionality as it calculates the positional information for each vertex, depending on where the camera is in the scene. For each call to this stage, vertex data (points of each polygon) are supplied as their local positions in the mesh. For this local position to be rendered in world view (within a relative coordinate system to the camera), the vertex stage takes the camera projection matrix and the model's world matrix and transforms them to their world positions.

The pixel shader handles colour and lighting within the scene. For this project, only a pixel-lit technique exists for calculating how much light is applied within the scene. Each pixel on the Window runs through this stage with information about the current polygon being drawn, and various point lights within the scene apply a lighting value for the current distance of the pixel's world coordinate.

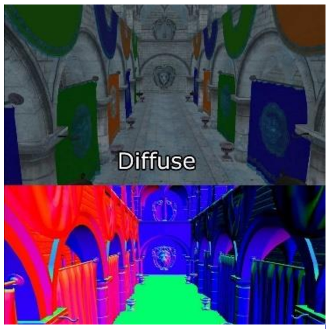


Figure 14 – Diffuse and Normal Textures (Van Oosten, 2015)

This stage also is supplied information such as the pixel's current world position, surface normal (or world normal for normal mapping) of the polygon, UV coordinates, and current projection position to the pixel shader. As seen in figure 14, both diffuse and normal data can be passed and evaluated as their own channels. This is important information for the pixel shader to evaluate the base texture, and the direction of the pixel to evaluate lighting and further advanced shading techniques. This information is enough for evaluating a pixel's current colour, and how much lighting should be received at the world position in relation to surrounding lights.

4.4 Scene Lighting

Forward rendering is the core render process used by this project. This is where each entity is drawn to the back buffer as a single texture, and the lighting is also calculated whilst the frame is being processed. This approach wouldn't have been appropriate in a larger scale project due to performance issues with large light sources, but because only a handful of lights were used (see figure 15 for sunlight example) this technique works best due to lighting being statically passed to the shader. Only when a light is updated does it need to be sent from machine memory to the GPU's memory, which can be a performance issue with large amounts of lights.

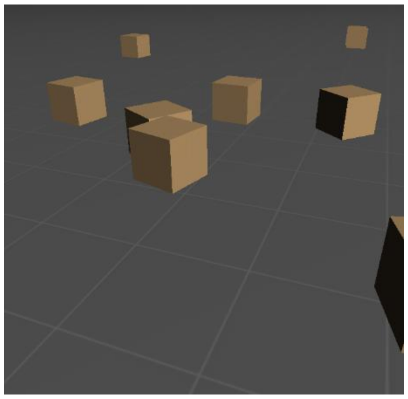


Figure 15 – Lighting applied in the project only affecting faces pointing aiming at the sun's position

Because the number of light sources is known at compile time, it was safe to assume that the shader file could be statically written to accept the number of lights used in the project. This gives to performance benefits such as loop unrolling, and prevents the surrounding issues with branching as there isn't any dynamic checks with if-statements within the shader code. Therefore, without these dynamic runtime expressions branch prediction can safely pre-evaluate future instructions whilst processing shader code. No costs are assumed as the shader just only must evaluate calculations passed as input, and no flow-control will throttle the rendering processes performance without complex pre-processing, see figure 16 (NVIDIA Corporation, 2005).

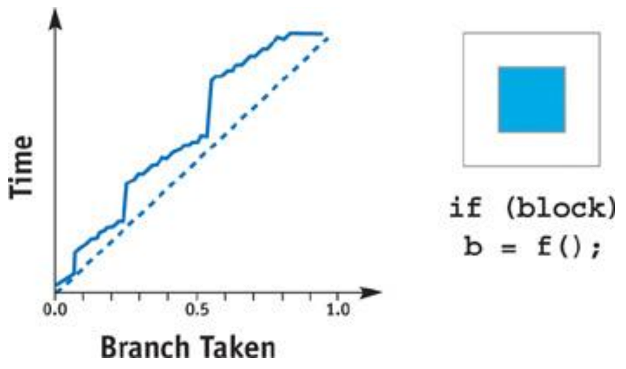


Figure 16 – Costs of branching within a pixel shader (NVIDIA Corporation, 2005)

4.5 Engine Interface

Throughout the application the graphics manager is accessed using an interface. IGraphicsManager declares standard operations for drawing geometry. A type of a graphics manager can then implement these functions making the accessing interface non-specific, making it possible to swap graphics managers without the worry of changing the project code. This project only uses a DirectX graphics manager, but it is possible to implement an OpenGL equivalent to target other platforms or to use alternative rendering techniques.

Because this project uses a DirectX approach for rendering geometry, it is divided into its own modular system hid behind the IGraphicsManager interface. The mesh importer is its own sub-system within the graphics manager, which can be accessed remotely from other systems (such as the asset provider) by invoking the graphics manager interface. This control of object ownership has circumvented the need for static members within the or global objects within the rendering engine. The only exception to this is the window manager which needs static initialisation for a Win32 specific message handler (related to backwards compatibility to C). Apart from that case, every sub-system and object instance within the rendering engine uses parent referencing to cross-communicate safely.

All object instances maintained by the rendering engine are explicitly shared ownership. Objects instances can be made with default shared_ptr initialisation, but they must reference a template and for them to stay in the scene they must do two operations:

1. **Register with the scene graph**

   Instances must be allocated to the scene graph. This is a call to

IScene::AddEntity, and this function will maintain a shared ownership copy of the object. If the scene is destroyed, the shared ownership will be lost and will either delete the object or lower its usage count for another section of the game to handle its destruction.

It's also important to note that this registration of objects is thread-safe. The call to AddEntity is wrapped in a critical section, only allowing one thread at a time to operate on the underlying entity list. This decision to make it thread safe is to allow the networking thread to insert new entities that are streamed from the server.

2. **Mark as persistent or keep a reference**

   Instances inserted into a scene graph are frequently cleaned up if their usage is only the scene graph and the entity isn't marked as persistent. This automatic clean up routine is a memory management feature of the scene graph, ensuring objects aren't lost within the application unless declaring they're meant to stay until the scene is notified. See listing 3 for an example of object persistence with the game's skybox.

```
std::shared_ptr<GameFloor> floor(new GameFloor(m_pFloorInstance));
floor->SetRenderMethod(pShaderMethod);
floor->SetPosition(0.0f, 0.0f, 0.0f);
floor->MarkPersistent();
AddEntity(floor);
```

Listing 3 – Entity registration within scene initialisation to insert a persistent GameFloor instance

4.6 Physics Controller

Every RiGameNode instance can take an internal controller component. RiController is an abstract class that supports updating parent RiGameNode logic as a sub-module that can be swapped or optionally not set.

In the implementation, cubes and scenery entities use an RiPhysicsController class to drive rigid body movement. Each time the node receives an update, the physics controller updates its parent with the next physics simulation coordinate information. Because the physics system is an independent sub-module of the game world, RiPhysicsController will only copy the current rigid body's state into the node. This is multithreaded approach to running physics independently, and it also makes physics processing faster as it doesn't have to traverse the world for rigid bodies to process.

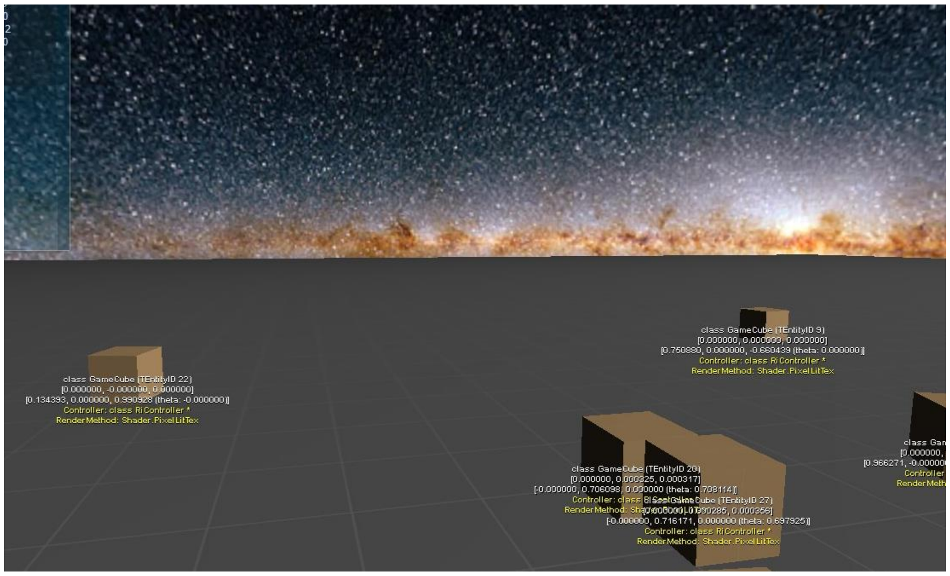
Along with updating the parent node coordinates, the physics controller also provides an interface for retrieving data from the rigid body, such as mass and velocity information for networked cubes.

This abstraction of controllers also allows for future module development, as any new class that derives from RiController can be added as a logic component (ex. AI module, destruction module).

4.7 Debug Overlay

Whilst developing and analysing game object instances, a debug overlay was implemented to show internal object data such as coordinate data and entity IDs. By pressing the tilde key, an overlay appears showing extended information.

As seen in figure 17, various information about the object types is displayed. RTTI (Run Time Type Information) is used to return the actual types of objects to show their internal usage within the code base. Along with this the controller type, if allocated, is also shown and so is the render method used to draw the entity with. This information is crucial for debugging the world state, and assisted development with viewing replicated entity's coordinate data without having to step through a debugger.



```
class GameCube (TEntityID 22)
[0.000000, -0.000000, 0.000000]
[0.134393, 0.000000, 0.990928 (theta: -0.000000)]
Controller: class RiController *
RenderMethod: Shader.PixelLitTex

class GameCube (TEntityID 9)
[0.000000, 0.000000, 0.000000]
[0.750880, 0.000000, -0.660439 (theta: 0.000000)]
Controller: class RiController *
RenderMethod: Shader.PixelLitTex

class GameCube (TEntityID 20)
[0.000000, 0.000325, 0.000317]
[-0.000000, 0.706098, 0.000000 (theta: 0.708114)]
Controller: class Ri
RenderMethod: Sha

class GameCube (TEntityID 27)
[0.000000, -0.000285, 0.000356]
[-0.000000, 0.716171, 0.000000 (theta: 0.697925)]
Controller: class RiController *
RenderMethod: Shader.PixelLitTex

class Gam
[0.000000,
[0.966271, -0.00000
Controller
RenderMeth
```

Figure 17 – Object instances with the debug overlay enabled

4.8 Summary

Overall the rendering engine has been developed to be modular and abstract enough for continued feature development. Its purpose is to render a dynamic set of objects whilst maintaining direct access to object information from other threads. This is an engine that can be built upon and be interfaced well enough that other systems can control specific instances within the system. This is demonstrated well in this project as the networking library has exclusive and safe control of 3D objects.

The reusability of this engine is also very appealing as there isn't any fixed or static typing that restricts this for one purpose.

5 Network Control

5.1 Introduction

The transmission of events is an important feature to have in a networked application. It cannot be solved with entity synchronisation, and sending parameters attached to events must have its own encoding for standard network transmission. This section covers Protocol Buffers, a universal encoding platform meant for remote procedure calling and the various other caveats of reliable transmission and object persistence.

5.2 Protocol Buffers

Protocol Buffers is an open source library developed by Google that parses a C-like syntax file into C++ classes to represent serialisable data. It's designed to be a fast, simple to use interface for exchanging raw data to and from a pre-defined structure. Most importantly it's robust as the project has been active since 2001 and has unit tests to ensure serialisation outcome is (Google Inc., n.d.).

In this project protocol buffers are used for RPC invocation with parameter data. Arguments sent with an event must use a pre-defined "protobuf" structure for the receiver to decode. Within the network library protobuf is used extensively for transferring key value pairs. Networked lists (an associative container) use these protobuf classes for transferring map state changes and even for transferring map values as streams.

5.3 Sequencing & Channels

Communication through network is split between reliable and unreliable calls. Unreliable calls in this project have no sequencing or extra special flags for how they are processed and sent. This is to ensure unreliable packets arrive at the destination with the highest speed without causing too many reliability issues. Reliable packets however can be given two extra flags for how they are queued and arrive to the target.

Sequenced packets are sent in order. They are not processed as they arrive but in order of how they were sent. With sequencing, it makes structured packet request arrive and process in a safely order. This project does not implement it, but a text chat system would depend on sequencing as the order of messages incoming should always be the order they were sent. If this wasn't the case, some chat messages may appear in the wrong order and confuse an active discussion.

One pitfall that is easy to fall into with sequencing is sending lots of packets continuously using these flags. If a stream of consistent data is all sequenced, it only takes one packet to be lost, retried or slowly processed by the receiver to cause a delay with the rest of the incoming data.

Because this problem can cause the entire network manager to lag until the infringing packet resolves, there's an extra way of dividing packets into sequenced "channels". Channels are uniquely identified codes that represent groups of sequenced data, where data streams may be divided into their own sequenced channel and other timely fashioned reliable data is allocated its own.

All packet operations based on IPacketFactory::CBasePacket provided an interface for setting these sequenced parameters on any outgoing call. Entity synchronisation does not allow sequencing as its usage with entity attributes is not appropriate as reliable network variables are merged into the same snapshot stream.

5.4 Remote Procedure Call Invocation

As previously discussed in this report's design section, incoming messages are routed depending on their message code. Within both projects, the server and the client, remote procedures take place to invoke functionality. This is almost identical to calling a function asynchronously except data must be encoded and sent to a client.

One of the major examples of usage of RPC in this project is the "spawn box" key. Pressing 'T' spawns a block and throws it in the direction of the player. An event is fired at the server and the server routes the event code to the appropriate listener.

This call is also bi-directional as servers and clients have a similar same interface for sending and receiving data. CProtoPacket<T> which contains and dispatches the protobuf packets also implements the base class CBasePacket for sequencing and channels.

5.5 Object Persistence

The networking library made for this project ensures the server is authoritative with entity creation and deletion. This model ensures that the server acknowledges client’s requests to make new entities (as seen with the ‘T’ key event) and so that the client doesn’t have full control to the world state.

The server however can give players control of entities within reason. This is heavily depended on in the project as when new players connect the server creates a new player object and allocates control to the player. Once the client acknowledges the new object they know what their local player entity is, and can start controlling it with asynchronous setting of data (such as the player’s current location).

Internal object ownership follows the same procedure as scene entities. When the server constructs a new object, the caller receives a reference to the object and shares ownership. At this point the object is shared between two sections of the program, the network entity manager and the caller. The caller can continue sharing the object, but once the usage count is only the network manager the instance is garbage collected by the network library. Upon deletion, all active players are also notified of the object being destroyed to ensure no objects continue being updated in the world state.

5.6 Summary

Network control is an important topic to cover when developing networked systems. The importance of connecting high-level operations to low-level transmission is high. Low-level operations should be robust and reliable as network control is connecting high level abstraction, and issues surrounding this may be detrimental to the overall product. Issues relating to this stage of online games can cause the following issues:

1. **Cheating**
   Object persistence and authority is an attack vector for cheaters to alter the state of the world without explicit permission. Improper RPC calls that use unsafe encoding could also be misused as buffer overflow attacks, which is what makes this topic important.
2. **Desynchronisation**
   Desynch happens when an expected event does not happen. RPC calls that

don’t arrive or are malformed will cause the event to be misinterpreted, desyncing the current networked environment for the receiver. Improper sequencing with channels can contribute to this problem, which the project already addresses with an interface.

Overall this topic is important because of its foundations to networking high-level abstraction. Object synchronisation and remote events cannot function without proper network control.

6 Entity Synchronisation

6.1 Introduction

Networking objects is usually in most applications approached with RPC calls and custom events for each data type. This usually leads to a lot of maintenance and updating core event handlers just for adding new variables that need networking. In this project, network variables are treated as their own instances, and are processed as a list of runtime attributes but still retain the static typing of class properties with encapsulation.

This section discusses the serialisation of networked changes, and the benefits this project has compared to a remote procedure based state update.

6.2 Serialisation of Deltas

To make bandwidth as small as possible, the networking library used for the project only sends changed data throughout the network update routine. New players are informed of new object states, but changes made throughout the world will only encode their changes into the next network snapshot instead of sending a full update of the world.

Classes that derive from "NetObject" receive includes to macro helpers which help build a networked class. Instead of a variable being declared as a standard property within a class, macros are supplied that can network non-complex data types. The NetworkVariable macro allows any set data type to be networked using a binary copy. This macro expands into an initialisation line which on construction of the object allocates it to an internal std::map within the object which lists all network variables assigned to an object. This runtime based variable listing is powerful for returning attributes of a base NetVar object, meaning the snapshot process doesn't need to know derived data type to return its data properties.

The only disadvantage to this approach is that only binary only data types work. "std::string" will not work due to its underlying container memory using heap allocation. There are also helpers such as NetworkString to assist in encoding fixed length strings.

Each network variable derives an abstract class called "NetVar" which introduces an interface for getting and setting the variables datatype. Setting data to a network variable invalidates it, and on the next snapshot pass it will be copied into a snapshot stream and marked as processed. When attempting to read from a network variable, the implementation can also return how many changes have happened since the last update with “GetChanges()”. This is useful for checking if a variable’s data has changed recently.

Alongside the network variable macros, there is also NetworkBuffered helpers which assist with replacing the accessor interface for the variable to support interpolated and extrapolated values for blended results.

This delta compression method also assists with making multithreaded changes to entity attributes with POD data types.

6.3 Multithreaded Data Manipulation

Within the project there are multiple threads taking various tasks. The main threads are the game render loop, which also processes game logic, and the networking thread. The update rate of the networking thread is fixed to a synchronisation rate or a tick rate, and this can be low compared to the demands of modern game FPS rates.

To make these mutually exclusive, the network sub-system runs independent from the rendering. This has a lot of benefits due to neither thread needing to wait on each other to complete tasks, and the network thread can run at its own rate without causing network rate complications. However, to make this possible there needs to be explicit access to object data to prevent multithreading issues with data access. This proves to be a problem when both systems share the same objects as remote players will be built on each client as a 3D model.

The first thing that makes this approach possible is variable invalidation. Manipulating a network variable from any thread will change the data, but it won’t invoke a sync update. Instead the caller invalidates the variable by increasing its invalidation count by one. When a snapshot pass happens, invalidated variables are checked and their data is serialised into the stream. After that the variable is then set back to its valid state. This is another reason why complex data types are not allowed as network variables, not only is the contents impossible to network, but variables should be simple enough that multiple threads can read and write without causing complications.

Another way this project manages communicate multithreaded sub-systems effectively is that the rendering thread will never touch the network object. Only RiGameNode classes can be rendered and updated, and NetObject is not inherited nor derived from any associated class to RiGameNode. Instead, NetObject's schedule RiGameNode instances to be made asynchronously, and then once the node is allocated then the network system sets renderable information by copy from the network variables. This enforces a strict one way communication policy where the network manager always retains active ownership of renderable instances.

6.4 Summary

Entity sync is an important sub-system topic to cover in any networked system. The relation of objects in a distributed environment relies on many serialisation techniques to ensure data is consistent and appropriate for the implementation. Multithreaded access not only improves performance, but supports the surrounding project of the networked library as there's little to no concern needed for critical sections or mutual exclusion locks to keep things thread safe. Having this approach from the start makes the library easier to interface with, which is mutually beneficial for both projects.

7 Replication

7.1 Introduction

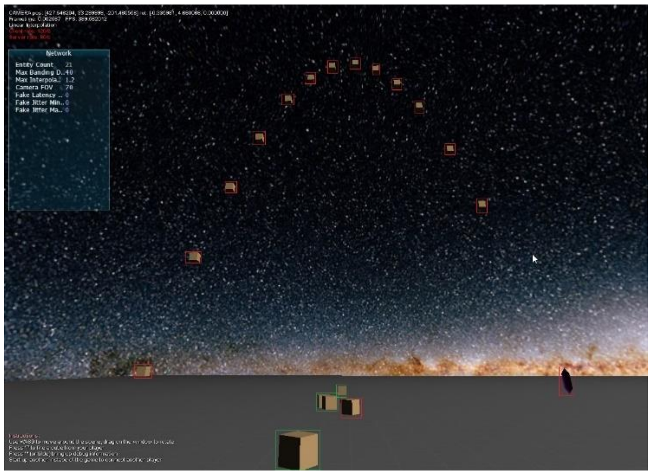
In this project, entities are not logically updated on the server. Instead the server acts as an arbiter connecting all players, processing RPC messages and handling authoritative control of creating entities and deleting them. Clients themselves send object states, attributes and events to other players. This is the act of replication, and simulating multiple entities owned by various connected players forms a completely player to player synced environment.

7.2 Object Ownership

Entities created by the server can be allocated an “owner”. A player can only replicate an entity when it’s locally controlled by themselves. This is an ownership level, and there are restrictions in place that prevent other players from attempting to change unowned objects on the server.

When a player receives a new instance from the server, they are informed of who controls it. Within the NetObject::NetworkUpdate() function, two functions are provided within the base class to determine whether the object is owned or not. IsNetworkLocal() returns true if the object is under the local player’s control, and the opposite happens for IsNetworkRemote().

Throughout the application these helper functions are used to determine whether to update certain replication routines. For example, the player will only update an instance of a PlayerController if the object is local and a controller exists. From there the local player will set the camera eye position to the entity position. If the player is remote, then the update call changes and instead it updates the RiGameNode to set the model position to the player position. This is the same for remote cubes, remote coordinates (position and angles) will only be applied to remote objects, see figure 18.


```
CAMERA pos: [427.546204, 93.289699, -201.480568] rot: [0.336861, 4.680086, 0.000000]
Frametime: 0.002687   FPS: 369.062012
Linear Interpolation
```

```
Network
Entity Count       21
Max Banding D..  40
Max Interpola..   1.2
Camera FOV        70
Fake Latency ..   0
Fake Jitter Min.. 0
Fake Jitter Ma..  0
```

Figure 18 – Replicated entities highlighted in red, green is locally owned and send changed data to red.

7.3 Physics

Running on each entity is a physics update based on rigid body dynamics. Bullet is a physics engine designed to interpret linear and angular velocity as force and torque. Mass is then factored in to complete the next step of a rigid body's motion.

Locally and remote objects run these physics calculations, regardless of ownership. Even though entity rotations and positions are interpolated, the physics engine can make good guesses into where the rigid body should be moving towards. In this project, linear velocity, angular velocity and mass is synchronised alongside positional data to support this idea of replicated physics.

The limitation to this approach is that when locally owned objects hit other player's objects, there is a momentary delay on the collision arriving on the remote player's screen and then the local player receiving the collision resolution (bouncing away, or not interacting). Even though this can cause some unpredictable results with multiple collisions/interactions being affected by latency, it is worth having it due to how cubes can be extrapolated a lot more correctly as torque can be predicted as well as linear movement.

7.4 Snapshotting

Each tick of the game server prepares two snapshot buffers with variable deltas to be sent to currently connected players. These buffers prove to be reliable and fast for data transmission due to their underlying implementation being pre-prepared for the low-level interface for sending buffers to players.

All network variables declared in classes are divided into two types, reliable and unreliable. Reliable network variables must arrive reliably without concern about speed. Unreliable is meant for fast transit, without reliability concerns. When a snapshot buffer is finalised, it must be copied into a low-level packet signifying its transmission parameters. Because there is a mixture of reliable and unreliable, the network library allocates two snapshot buffers and processes them both with only their associated network variables.

Snapshots are contained as SnapshotStream class types, and their only construction parameter is an optional snapshot buffer size. When the object is built, it allocates a predefined amount of memory to use for storing snapshot data. The snapshot stream will then exist as a composition of the network manager for reuse. When data needs to be appended to the snapshot buffer, the library calls SnapshotStream::AllocatePage with the amount of bytes that need to be reserved. This returns a Page object with bounds checking, and ensures that any data wrote to the Page will never go outside its allocated buffer.

Once the buffer has been prepared, a packet is made to be sent to players with the snapshot buffer's starting address and how many bytes were allocated.

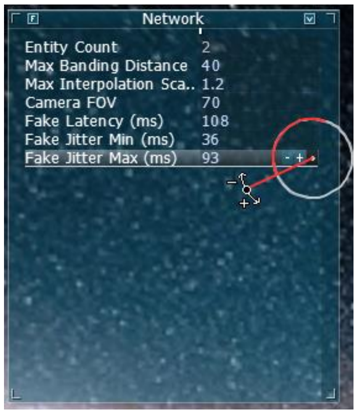
Apart from the heap allocation at the start, there is no further dynamic allocation after a snapshot buffer is initialised. This makes compositions of snapshot pages as fast as incrementing the current position and returning a lightweight object (or nothing for the Allocate function) to represent a section of memory. See listing 4 for an example of the network manager allocating a header frame to the passed snapshot buffer.

```
SnapshotHeaderFrame* NetSystem::AllocateSnapshotHeader(SnapshotStream& stream)
{
	std::unique_ptr<SnapshotStream::Page> page =
		stream.AllocatePage(sizeof(SnapshotHeaderFrame));

	 SnapshotHeaderFrame* header = page->GetBlock<SnapshotHeaderFrame*>();

	// Snapshots are give a special reserved network code to route
	// to snapshot handling code.
	header->code = NETID_Reserved::RTTI_Object::OBJECT_FRAME;
	header->frame_count = 0;
```

```
        return header;
}
```

Listing 4 – [NetSystem.cpp] Snapshot header allocator utility function

7.5 Latency Module



Network

```
Entity Count               2
Max Banding Distance      40
Max Interpolation Sca..  1.2
Camera FOV                70
Fake Latency (ms)        108
Fake Jitter Min (ms)      36
Fake Jitter Max (ms)      93
```

Figure 19 – Latency UI Helper

To assist with testing, the network library also contains a small sub-system called NetFakeLatency. This module can be hooked into the network manager as a proxy for outgoing packets to apply various latency variables.

The module runs an independent worker thread that waits for jobs to process scheduled outgoing packets. When an outgoing packet is proxied into the module, it is sent to the scheduled list and is applied parameter data to simulate a fixed latency value and/or jitter.

If the jitter max and min parameters are valid, a random delay between the two values in milliseconds is applied to the target outgoing time. If the fixed latency value is higher than zero, the target outgoing time also adds that on.

Once the worker thread validates that the current time has surpassed the outgoing time of the packet, it is then dispatched to the network system which demonstrates latency issues for testing.

The latency module parameters are currently controlled in the implementation by a user interface component. This UI component is an AntTweakBar window that gives direct access to the latency parameters as seen in figure 19.

7.6 Summary

Within this program object replication is a key concept that drives most gameplay. Objects with local updates should broadcast their states to other players, which is what this library accomplishes. It is incredibly flexible to define new network objects and entity managers and to get parameters synced due to the replication platform only needing knowledge of the NetObject and its containing data. Because these modules co-exist, appropriate multiplayer gameplay can be achieved without showing significant signs of network latency. This is what replication tries to achieve, single player objects but networked.

8 Test Strategy

8.1 Introduction

This project relies on multiple instance communication, meaning two instance of the game must connect to one server. Testing this will require simulating events such as jitter and latency with individual clients. The test strategy will cover network performance, CPU usage and memory allocation for networking. This section will also cover client performance, which is individual frame rates in various configurations (debug and release), with a fixed object count and with debug overlay text enabled. The look and feel of the overall product will also be tested to evaluate the visual quality of the product.

8.2 Game Performance

Games by design must be developed with priority to real time processing. The count of how many frames per second a game runs at can lower the value significantly if the rate is too choppy. User experience is important; therefore, frames per second and overall visual quality are the primary focus of this project’s test strategy.

The algorithms and modular design have benefitted not only game features, but also the experience the user receives. Modular components and instance templating has significantly reduced the total amount of memory consumed throughout the application.

Configuration	**CPU% Usage**	**Memory (MB)**	**#Threads**	**#Cores Available**	**FPS**	**Test Case**
Release, client	29%	44.4	18	8	600	0 local ents 0 remote ents
Release, client	28%	43.9	21	8	644	10 local ents 0 remote ents
Release, client	29%	45.3	21	8	547	10 local ents 10 remote ents
Debug, client	31%	56.8	22	8	488	0 local ents, 0 remote ents
Debug, client	29%	58.6	19 threads	8	714	10 local ents 0 remote ents
Debug, client	31%	57.8	21 threads	8	288	10 local ents 10 remote ents

Release, client	27%	44.5	21 threads	8	542	55 local ents 55 remote ents
Debug, client	35%	57.8	19 threads		55	55 local ents 55 remote ents

Table 1 – Performance tests of various world states

As seen in table 1, memory usage in release builds scales upwards at a very slow trend. Entity instancing has meant a lot of shared resources are reused, so the memory footprint of a large environment in this application is miniscule compared to the actual requirements (rendering a mesh for each entity).

Another steady trend in the results table is CPU usage increasing as more entities are introduced into the environment. This highlights the importance of multithreaded sub-systems as having multiple cores to process on can make the overall product scalar. The need for more processing power is not bottlenecked with other systems as in this project both the rendering system and the network system have their own processor cores to themselves, giving freedom to use more CPU time.

The table also shows a clear drop in performance with the debug configuration. This is to be expected as debug applications are usually unoptimised to benefit the debugger, whereas release mode does lots of instruction optimisation at compilation. This reflects with the frame count of release modes being significantly higher as certain libraries (such as the Microsoft Standard Template Library) are optimised thoroughly for production software.

8.3 Latency Spoofing

The networking library for this project has a built in fake latency module that can change the outcome of packet timing in real time. This module was used to spoof various network environments as the control of packet latency and jitter helped with continuously improving the interpolation techniques.

Jitter is networking is the randomness of packet dispatching. Packets are expected to arrive as fast as they can, but jitter on a network could cause unstable results where packets arrive at random intervals and sometimes out of sequence.

Applying fake jitter to clients gave give good results to how an unreliable connection would cope attempting to simulate and replicate entities. The results of this showed that replicated entities of players with jitter would make their objects lag around the world, even if other players had a stable connection to the server. Interpolation and extrapolation tried to correct the visible randomness of packets, but it would never be able to predict packets that arrived randomly out of sequence. This gave a good understanding of what latency can do to an online game, and these issues gave a good point towards why entity replication may not be a good thing for gameplay that depends on the responsiveness of a simulation.

8.4 Bug Reporting

Whilst testing this project with multiple instances on various machines, catching bugs in release mode is hard without a proper minidumping library. Distributing the game as a debug release wouldn’t be appropriate either as the program would be unoptimised and depend on the machine having Visual Studio to run the debug Visual C++ libraries.

Instead this project uses a sub-system independent from the rendering and network library for logging events to file. The debug logger (DLog) system makes globally accessible functions for reporting various information to the console window and to file.

Because the logging system is multithreaded, if the rendering or network threads crash, the logging thread will still exist to write any last-minute messages to file before closing the game. For critical asserts where the game must quickly shut down, the logging system also provides a terminate function for logging important information then instantly shutting down all running threads. This has proved useful for conditions in the application that should not fail, and to end the game before any undefined behaviour takes place.

8.5 Summary

Overall the project has scored well for creating an interactive environment without affecting the project’s value. The frame rates produced in the test results have shown that even with over a hundred active entities in-game, the frame rate is still way above average. A lot of this has come down to the splitting of systems across multiple threads, as it has shown it has mutually benefited each module by isolating CPU consumption.

The testing strategy for this project has been a success as an appropriate framework has supported various techniques, such as logging to file and network latency spoofing.

9 Evaluation, Conclusions and Future Work

9.1 Project Objectives

This project started as a demonstration of entity replication in first person shooters. The outcome I was expecting with my original literature findings, compared to what was delivered in the end, has given me a bigger insight to the complications and caveats that surround online games.

My original intentions were to create a shooter where the environment would be replicated locally and by other players, removing the need for a dedicated server to handle gameplay logic. The idea of having clients replicate a simulated world was something that I wanted to prove could be done. The surrounding research behind this is very niche and I wanted this project to be a solid demonstration that it works, and can be integrated into games just as well as alternative methods.

This project has shown that the techniques identified in the literature review can be executed, and even though it didn't get to the stage of being an interactive shooter as I had originally planned, it demonstrated and produced technology to make it possible.

9.2 Evaluation

This project has covered a broad range of technology used in video games, and I'm very happy with the final product. As previously mentioned, I wanted to get a fully functioning first person shooter game using replicated technology. These goals were ambitious, but got this project far enough to develop a technology demonstration of what's needed in online games with entity replication.

I managed to get object instances and object types networked using effective serialisation, which is a lot further than I had originally planned in section 1.2. Not only could players replicate entities, but this usage proved to be useful for player objects and even sending physics data. Alongside this I also did extensive research into interpolation and extrapolation methods with coordinates and angles, including the usage of quaternions for representing angles.

In hindsight, I would have stuck to just building a tech demo, and focused my work primarily on the network library. This would have given me more time to develop on the networking engine, and to focus primarily on bringing alternative techniques alongside my research to compared the advantages and disadvantages of each. This could have been comparing snapshotting to direct entity updating, and could have helped demonstrate the requirements of online games.

But I know that if I had stuck to just the networking engine and my implementation on top of it wasn't to the standard that it currently is, I wouldn't have invested time into making the networking portable and flexible. Because I originally had the idea of a shooter game as my product, I designed the internal interface of my network engine to be flexible and compatible for future usage or if my specification changed. This approach was also taken on my rendering implementation as the interfaces and structure of that system is also flexible enough that both engines can exist independently without needing each other to function.

In regards to entertainment value, this project doesn't have a lot of gameplay elements, which is on purpose. This project is a tech demo showing entity interaction across the network using entity replication. There is a future possibility that this work is eventually is extended into a game with entertainment purpose, as the rendering engine and network engine has documented and easy to understand interfaces for extending and adding new functionality.

9.3 Applicability of Findings to the Commercial World

The project and research I've made for this project has potential commercial value as online gaming is becoming one of the biggest sources of revenue for studios (Ahmed, 2017). These findings and techniques demonstrate how online games can communicate by using clients as hosts, but also how alternative variants would be better such as peer-to-peer connectivity.

Game studios and independent developers are always looking for multiplayer solutions, this networking library operates as a standalone interface for connecting abstract data types online. Its integration process and features makes it appealing for future usage in the commercial world.

9.4 Conclusions

With this project, I have understood and considered common approaches to networking online games, and demonstrated the advantages and disadvantages by introducing basic gameplay elements such as physics and player models. The time invested into research gave a good foundation for evaluation of methods, and the corresponding implementation showed that the current research on this topic is valid and should be considered by professional developers.

Not only does this project take other work into account, it itself is a product that can be used by others. I'm proud of the resulting project as it solves the problems I had originally wrote about in this report, and it's something I can carry on to other projects as a platform for online games.

9.5 Future Work

Considering my final product, I can start to see the disadvantages to this approach that I wouldn't have thought about at the start of development. Replicated entities must communicate states from player to server and to other players. I found that this amount of sending and receiving can add up to 150 milliseconds to a synchronisation event. If a thrown block hit another player's block, it wouldn't be an instantaneous bounce or impulse. Because of the communication to and from the server, collisions and interaction in-game would be slightly delayed making latency look obvious. An example can be seen in figure 20, where an entity is sliding into another entity. This registration from player red from green with information about the collision would have to relay through the server, causing a lengthy delay.

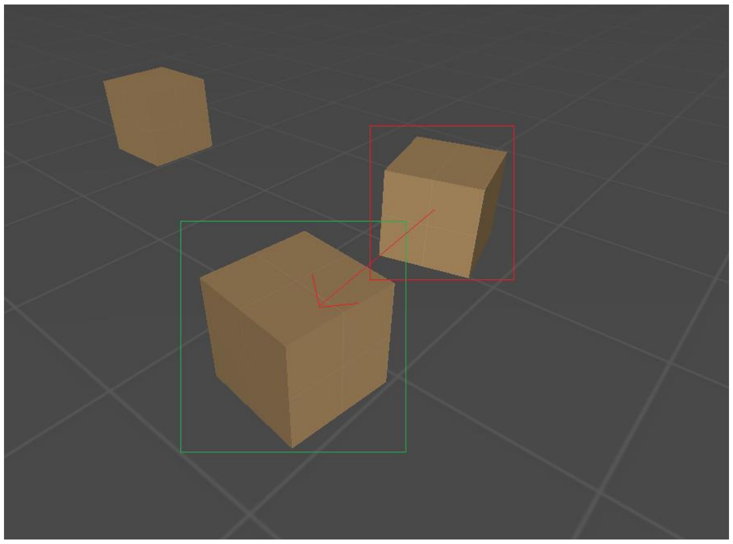


Figure 20 – An example scenario where it would take ~150ms for a hit to be registered on the remote player's machine, and for it to be sent back

Looking back, I would have approached network communication as peer-to-peer instead of a dedicated server acting as a relay. As previously mentioned in my literature review, there is a lot of usage of peer to peer in commercial games. It induces security risks, but gives direct communication between players. This induced latency on collision events take a third of the time to arrive as a server does not have to rebroadcast the data.

Another thing that I feel this project lacked was ownership migration. When players left the server, their spawned objects would be destroyed as their ownership expired. If I had more time to work on this in the future, I would have implemented ownership migration so that objects could have been carried to other players depending on certain circumstances, such as a player being too far away or a player disconnecting.

Whilst planning my test strategy at the start of the project, I had hoped to implement a network graph user interface. Because of the short amount of time I had to focus on the actual game and networking engine, I had to drop this interface to save time. Information such as latency and update statistics could have helped with testing the product overall as in depth details about the networking engine weren't exposed to be analysed.

The rendering engine itself was also limited to basic rendering techniques which could have been extended on. The world environment could have been more appealing if I had spent time on bringing more models and shader techniques into the game. This is something that could be developed in the future as the engine itself was designed to be easily extended.

9.6 Concluding Reflections

I didn't fulfil my initial plan with this project as I had to cut out some gameplay to focus more on the actual framework, but the outcome of developing this has been successful and has brought a lot of research and information into light.

The final product that came out of this project is something is almost exact to what I was hoping to achieve. The methodologies and engine related work needed goes further than what I had originally planned. Covering low-level networking with UDP and TCP, to building interfaces between protocols and high-level object abstraction has given this project large amount of topic coverage. This project and report is something I can look back at for future networking reference, as the research and implementation doesn't just describe methodologies but demonstrates them in a real-world example. I hope to continue with development with this project by extending the networking engine further, and hopefully pursue the topics covered in this report in the future.

References

Ackermann, J. (2012) 'Playing Computer Games as Social Interaction: An Analysis of LAN Parties', in Fromme, J. & Unger, A. (eds) *Computer Games and New Media Cultures,* s.l.:Springer Netherlands, pp. 465-476.

Ahmed, H. (2017) *GTA Online Revenue Reaches $500 Million,* http://segmentnext.com/2016/04/13/gta-online-revenue-reaches-500-million/ (accessed 17 March 2017).

Aldridge, D. (2011) *I Shot You First! - Gameplay Networking in Halo: Reach,* s.l.: Bungie, Inc.

Anand, B., Sebastian, J. & Ming, S. Y. (2011) *'PGTP: Power aware game transport protocol for multi-player mobile games',* s.l., IEEE.

Baughman, N. E. & Levine, B. N. (2001) *'Cheat-Proof Playout for Centralized and Distributed Online Games',* s.l., IEEE.

Bernier, Y. W. (2001) *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization,* https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization#Footnotes (accessed 18 March 2017).

Dam, E., Koch, M. & Lillholm, M. (1998) *Quaternions, interpolation and animation (Vol. 2),* s.l.: Datalogisk Institut, Københavns Universitet.

Diehl, M. (2011) 'Network Programming with ENet', *Linux Journal,* 2011(210).

Eberly, D. H. (2006) *'3D Game Engine Design, Second Edition: A Practical Approach to Real-Time Computer Graphics',* 2nd edition, San Francisco, CA: Morgan Kaufmann Publishers Inc..

Epic Games, Inc, (2015) *Replication,* https://wiki.unrealengine.com/Replication#Concepts (accessed 17 March 2017).

Färber, J. (2004) 'Traffic Modelling for Fast Action Network Games', *Multimedia Tools and Applications,* 23(1), pp. 31-46.

Fiedler, G. (2008) *UDP vs. TCP,* http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/ (accessed 20 March 2017).

Fiedler, G. (2014) *Networked Physics - Snapshots and Interpolation,* http://gafferongames.com/networked-physics/snapshots-and-interpolation/ (accessed 18 March 2017).

Frohnmayer, M. & Gift, T. (1999) *The TRIBES Engine Networking Model,* s.l.: GDC.

Funkhouser, T. A. (1995) 'RING: a client-server system for multi-user virtual environments', *I3D '95 Proceedings of the 1995 symposium on Interactive 3D graphics,* pp. 85-92.

Google Inc., n.d. *Frequently Asked Questions,*
https://developers.google.com/protocol-buffers/docs/faq
(accessed 18 April 2017).

Guzman, A. O. D. n.d. *Unofficial Quake Network Protocol Specs v1.01b,*
http://www.gamers.org/dEngine/quake/QDP/qnp.html
(accessed 19 March 2017).

Jenkins Software LLC, n.d. *RakNet - Installation,*
http://www.raknet.net/raknet/manual/introduction.html
(accessed 20 March 2017).

Jenkins Software LLC, n.d. *RakNet - Supported Platforms,*
http://www.jenkinssoftware.com/platforms.html
(accessed 20 March 2017).

Lee, C. H., Baset, S. A. & Schulzrinne, H. n.d. *TCP over UDP,* s.l.: s.n.

Madhav, S. & Glazer, J. (2015) in *Multiplayer Game Programming: Architecting Networked Games,* s.l.:Addison Wesley, p. 2.

Microsoft, n.d. *Graphics Pipeline,*
https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx
(accessed 18 April 2017).

newzoo, (2016) *2016 Global Games Market Report,* s.l.: newzoo.

NTT Communications, (2010) *The Evolution of Ethernet,* s.l.: s.n.

NVIDIA Corporation, (2005) 'Chapter 34. GPU Flow-Control Idioms', in *GPU Gems 2,* s.l.:Addison Wesley.

OculusVR, (2014) *RakNet,*
https://github.com/OculusVR/RakNet
(accessed 20 March 2017).

Pantel, L. & Wolf, L. C. (2002) 'On the Suitablity of Dead Reckoning Schemes for Games', *NetGames '02 Proceedings of the 1st workshop on Network and system support for games,* Volume 84, p. 79.

Parziale, L. et al. (2006) *'TCP/IP Tutorial and Technical Overview',* s.l.:Vervante.

Phillips, T. (2014) *50 EA games will have their servers shut down,*
http://www.eurogamer.net/articles/2014-05-12-50-ea-games-will-have-their-servers-shut-down
(accessed 17 March 2017).

Postel, J. (1980) *User Datagram Protocol, RFC 768,*
https://www.rfc-editor.org/rfc/pdfrfc/rfc768.txt.pdf
(accessed 19 March 2017).

Roiser, S. (2003) *Reflection in C++,* s.l.: CERN EP/LBC, TU Vienna.

Salzman, L. (2015) *ENet: ENet,*
http://enet.bespin.org/index.html
(accessed 20 March 2017).

Sawashima, H., Yoshiaki, H., Sunahara, H. & Oie, Y. (1997) *Characteristics of UDP Packet Loss: Effect of TCP Traffic,* s.l.: s.n.

Shoemake, K. (1985) *Animating rotation with quaternion curves,* New York: ACM.

Smed, J., Kaukoranta, T. & Hakonen, H. (2002) *'Aspects of Networking in Multiplayer Computer Games',* Hong Kong, s.n., pp. 1-5.

Suznjevic, M., Saldana, J., Matijasevic, M. & Fernandez, N. (2014) 'Analyzing the Effect of TCP and Server Population on Massively Multiplayer Games', *International Journal of Computer Games Technology,* Volume 2014, p. 17 pages.

Terrano, M. & Bettner, P. (2001) *1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond,* s.l.: Gamasutra.

treeform (2016) *Don't use Lockstep in RTS games,*
https://medium.com/@treeform/dont-use-lockstep-in-rts-games-b40f3dd6fddb#.nh8gleiql
(accessed 25 March 2017).

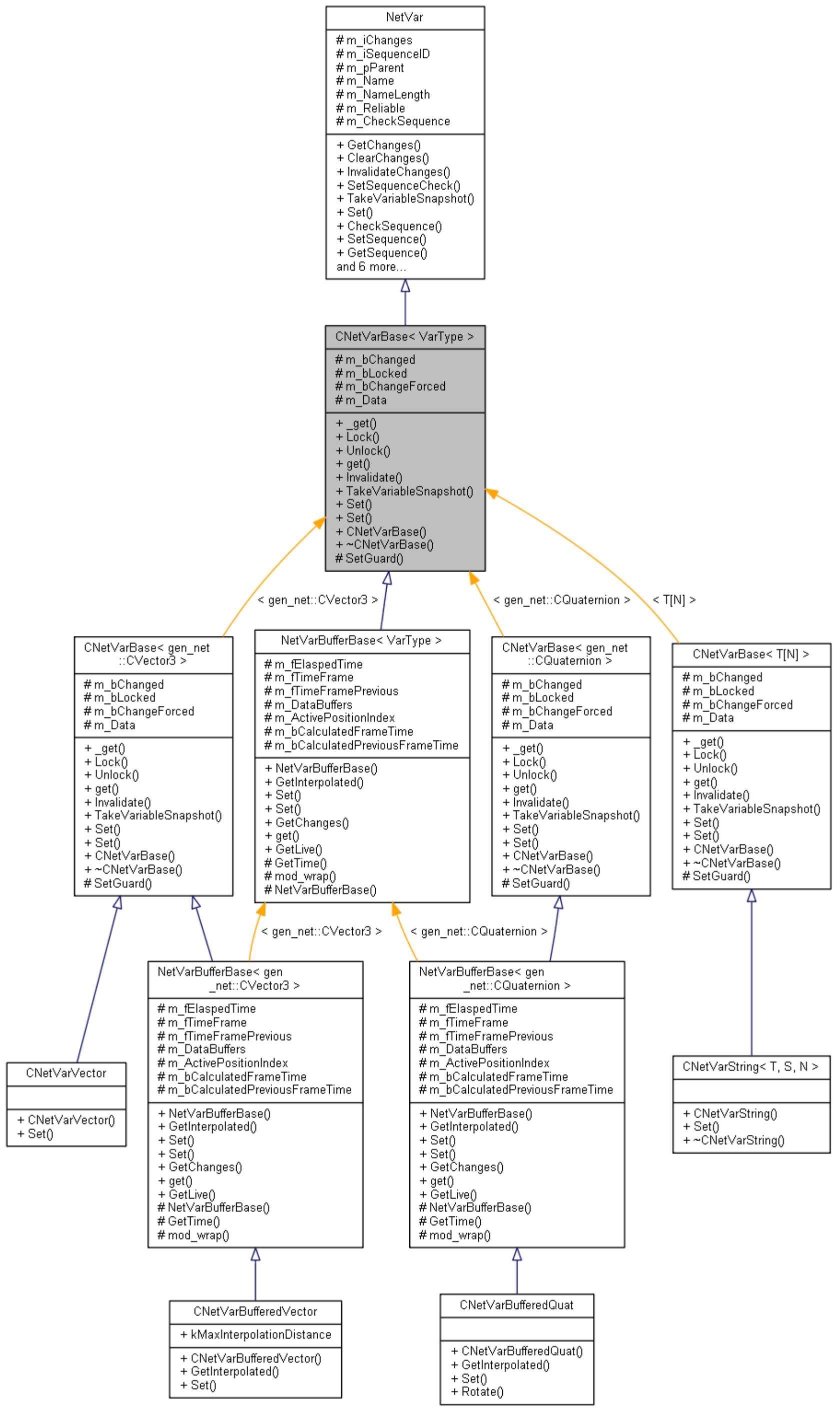
Valve Corporation, (2005) *Source Multiplayer Networking,*
https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
(accessed 19 March 2017).

Van Oosten, J., (2015) *'Forward vs Deferred vs Forward+ Rendering with DirectX 11',* [Art].

Wu, D. n.d. *Measurement Based Multiplayer Gaming Performance Study over Mobile Networks, master thesis,* s.l.: IMIT, Royal Institute of Technology .

Yin-Poole, W. (2017) *For Honor: Ubisoft answers the big questions,*
http://www.eurogamer.net/articles/2017-03-03-for-honor-ubisoft-answers-the-big-questions
(accessed 20 March 2017).

Appendix 1:
Network Variable Class Diagram for POD and non-POD Types


NetVar
# m_iChanges
# m_iSequenceID
# m_pParent
# m_Name
# m_NameLength
# m_Reliable
# m_CheckSequence
+ GetChanges()
+ ClearChanges()
+ InvalidateChanges()
+ SetSequenceCheck()
+ TakeVariableSnapshot()
+ Set()
+ CheckSequence()
+ SetSequence()
+ GetSequence()
and 6 more...
CNetVarBase< VarType >
# m_bChanged
# m_bLocked
# m_bChangeForced
# m_Data
+ _get()
+ Lock()
+ Unlock()
+ get()
+ Invalidate()
+ TakeVariableSnapshot()
+ Set()
+ Set()
+ CNetVarBase()
+ ~CNetVarBase()
# SetGuard()
< gen_net::CVector3 >
< gen_net::CQuaternion >
< T[N] >
CNetVarBase< gen_net ::CVector3 >
# m_bChanged
# m_bLocked
# m_bChangeForced
# m_Data
+ _get()
+ Lock()
+ Unlock()
+ get()
+ Invalidate()
+ TakeVariableSnapshot()
+ Set()
+ Set()
+ CNetVarBase()
+ ~CNetVarBase()
# SetGuard()
NetVarBufferBase< VarType >
# m_fElaspedTime
# m_fTimeFrame
# m_fTimeFramePrevious
# m_DataBuffers
# m_ActivePositionIndex
# m_bCalculatedFrameTime
# m_bCalculatedPreviousFrameTime
+ NetVarBufferBase()
+ GetInterpolated()
+ Set()
+ Set()
+ GetChanges()
+ get()
+ GetLive()
# GetTime()
# mod_wrap()
# NetVarBufferBase()
CNetVarBase< gen_net ::CQuaternion >
# m_bChanged
# m_bLocked
# m_bChangeForced
# m_Data
+ _get()
+ Lock()
+ Unlock()
+ get()
+ Invalidate()
+ TakeVariableSnapshot()
+ Set()
+ Set()
+ CNetVarBase()
+ ~CNetVarBase()
# SetGuard()
CNetVarBase< T[N] >
# m_bChanged
# m_bLocked
# m_bChangeForced
# m_Data
+ _get()
+ Lock()
+ Unlock()
+ get()
+ Invalidate()
+ TakeVariableSnapshot()
+ Set()
+ Set()
+ CNetVarBase()
+ ~CNetVarBase()
# SetGuard()
< gen_net::CVector3 >
< gen_net::CQuaternion >
CNetVarVector
+ CNetVarVector()
+ Set()
NetVarBufferBase< gen _net::CVector3 >
# m_fElaspedTime
# m_fTimeFrame
# m_fTimeFramePrevious
# m_DataBuffers
# m_ActivePositionIndex
# m_bCalculatedFrameTime
# m_bCalculatedPreviousFrameTime
+ NetVarBufferBase()
+ GetInterpolated()
+ Set()
+ Set()
+ GetChanges()
+ get()
+ GetLive()
# NetVarBufferBase()
# GetTime()
# mod_wrap()
NetVarBufferBase< gen _net::CQuaternion >
# m_fElaspedTime
# m_fTimeFrame
# m_fTimeFramePrevious
# m_DataBuffers
# m_ActivePositionIndex
# m_bCalculatedFrameTime
# m_bCalculatedPreviousFrameTime
+ NetVarBufferBase()
+ GetInterpolated()
+ Set()
+ Set()
+ GetChanges()
+ get()
+ GetLive()
# NetVarBufferBase()
# GetTime()
# mod_wrap()
CNetVarString< T, S, N >
+ CNetVarString()
+ Set()
+ ~CNetVarString()
CNetVarBufferedVector
+ kMaxInterpolationDistance
+ CNetVarBufferedVector()
+ GetInterpolated()
+ Set()
CNetVarBufferedQuat
+ CNetVarBufferedQuat()
+ GetInterpolated()
+ Set()
+ Rotate()