Multiplayer Networking in a Modern Game Engine

Alan Edwardes

i. Abstract

The idea behind this project was to create a prototype multiplayer networked engine, creating an engine platform to compare modern networking technologies and techniques.

Thorough research into different tools and technologies was performed, including analysing existing products on the market. Interviews were also performed, informing the project using the experience and expertise of knowledgeable people and industry professionals.

The final design of the prototype was carefully constructed using the research and project objectives, and the prototype was implemented using selected technologies.

The final prototype includes a mix of technologies, with implementations of extra systems in order to provide a realistic basis for testing and benchmarking. Analysis was then performed using the prototype, and low-level performance and networking monitoring tools.

Using the results of this analysis, and taking into account research performed in the project, it was found that a shift toward adaptive solutions was sensible, rather than deciding on any one technology or technique.

ii. Acknowledgements

I would like to express a special thanks to my supervisor, Dr. Keith Burley for his continual support throughout the project. I would also like to thank my friends and family for their constant support and encouragement.

iii. Table of Contents

1 Introduction 1

1.1 Preamble 1

1.2 Aim 1

1.3 Objectives 1

1.3.1 Research and Compare Existing Software and Development Methods 1

1.3.2 Compare Each Method and Tool, Evaluating Drawbacks and Merits 2

1.3.3 Decide Which Tool to Use, and How to Use it 2

1.3.4 Conduct Interviews With Knowledgeable People 2

1.3.5 Create an Engine Design Based on Existing Technologies 2

1.3.6 Implement, Test and Document a Multiplayer Game Engine Concept 2

1.3.7 Consider Different User Limitations Throughout, Such as Packet Loss and Latency 3

1.3.8 Evaluate Quality and Usefulness of the Application 3

1.3.9 Conclusions, Recommend Further Work 3

1.3.10 Critically Evaluate the Project 3

1.4 Task Plan to Complete Objectives 3

2 Tools, Technologies and Methodologies 1

2.1 Programming Languages 1

2.1.1 C♯ 1

2.1.2 C++ 2

2.1.3 Python with CPython 2

2.1.4 Java 3

2.1.5 Language Comparison 3

2.1.6 Language Selection 5

2.2 Graphics, Input and Sound Libraries 5

2.2.1 Graphics Technologies 5

2.2.2 SFML 6

2.2.3 Allegro 7

2.2.4 Qt 8

2.2.5 Library Comparison 9

2.2.6 Library Functional Analysis Score Table 9

2.3 Communication Protocols 10

2.3.1 TCP/IP 10

2.3.2 UDP/IP 10
2.3.3 DCCP/IP 11
2.3.4 Protocol Comparison 11
2.4 Physics Libraries 11
2.4.1 Havok Physics 12
2.4.2 Box2D 12
2.4.3 Physics Library Comparison 12
2.5 Similar Works 13
2.5.1 Source Engine 13
2.5.2 Torque2D 15
2.6 Development Methodologies 16
2.6.1 Scrum 16
2.6.2 Agile 16
2.6.3 Rapid Application Development 17
2.6.4 Methodology Comparison 17
2.7 Impact on Design 17
3 Research Approach 18
3.1 Forms of Research 18
3.1.1 Inductive or Deductive Reasoning 18
3.2 Data Collection Methods 19
3.2.1 Questionnaires 19
3.2.2 Interviews 19
3.2.3 Participant Observation 20
3.2.4 Case Study 21
3.3 Evaluation of Data Collection Methods 21
3.4 Research Approach Choice 21
3.5 Impact on the Project 21
4 Informing the Design 23
4.1 Areas that Require Research 23
4.1.1 Handling Dependencies 23
4.1.2 Language Decision 23
4.1.3 Network Packet Size 24
4.1.4 Network Packet Frequency 24
4.2 Interviewees 24

4.3 Research Results 25
4.3.1 Question 1 Analysis 25
4.3.2 Question 2 Analysis 25
4.3.3 Question 3 Analysis 26
4.3.4 Question 4 Analysis 26
4.3.5 Question 1 Conclusions 26
4.3.6 Question 2 Conclusions 27
4.3.7 Question 3 Conclusions 27
4.3.8 Question 4 Conclusions 27
4.4 Interview Conclusions 27
4.5 Impact on Design 27
5 Design & Development 29
5.1 Version Control 29
5.2 Build System 29
5.3 Development Environment 29
5.4 Self-Documenting Code 29
5.5 Designing an Engine 29
5.5.1 Class Structure 30
5.5.2 Shared Code 30
5.5.3 Client Library 31
5.5.4 Server Library 32
5.5.5 Network Code 32
5.5.6 Catering for Packet Loss 33
5.5.7 Main Loop 33
5.5.8 Configuration Files 34
5.6 Add Unit Tests 36
5.6.1 Areas Requiring Unit Testing 36
5.7 Barebones Code Structure 37
5.8 Fleshing Out The Code Structure 37
5.9 Adding a Basic Renderer 37
5.9.1 Structure 37
5.9.2 Implementation 37
5.10 Testing the Prototype and Fixing Bugs 38
5.10.1 Continual Testing 38

5.10.2 Assertions 38
6 Implementation & Testing 39
6.1 Unit Testing 39
6.2 Performance Analysis 39
6.2.1 Hypothesis 39
6.2.2 Data 39
6.3 Network Packet Tracing 40
6.4 Dealing with Firewalls 40
6.5 Configuration 42
6.6 Viability of TCP/IP and UDP/IP 43
6.6.1 Nature of the Game 43
6.6.2 Considerations for TCP/IP 43
6.6.3 Considerations for UDP/IP 43
6.6.4 A Hybrid Approach 43
7 Evaluation 45
7.1 Existing Software, Methods and Tools 45
7.1.1 Research and Compare Existing Software and Development Methods 45
7.1.2 Compare Each Method and Tool, Evaluating Drawbacks and Merits 45
7.1.3 Decide Which Tool to Use, and How to Use it 46
7.2 Conduct Interviews With Knowledgeable People 46
7.3 Create an Engine Design Based on Existing Technologies 47
7.4 Implement, Test and Document a Multiplayer Game Engine Concept 47
7.5 Consider Different User Limitations Throughout, Such as Packet Loss and Latency 48
8 Conclusion 49
8.1 Summary 49
8.2 Recommendations 49
8.3 Further Work 50
9 Table of Figures i
10 Table of Tables i
11 References ii
Appendix 1 Project Specification i
Appendix 2 Project Schedule xii
Appendix 3 Test Code Reference Rendering xiii
Appendix 4 Pure OpenGL Implementation of Reference Rendering xiv

Appendix 5 SFML Implementation of Reference Rendering xv
Appendix 6 Allegro Implementation of Reference Rendering xvi
Appendix 7 Qt Implementation of Reference Rendering xvii
Appendix 8 Source Engine Server and Client Class Diagram xviii
Appendix 9 Source Engine Server Class for a Dynamic Light xix
Appendix 10 Source Engine Client Class for a Dynamic Light xx
Appendix 11 Interviewee Background Information and Consent Form xxi
Appendix 12 Interview Results xxii
Appendix 13 Shared Code Class Diagram xxv
Appendix 14 Client Library Class Diagram xxvi
Appendix 15 Server Library Class Diagram xxvi
Appendix 16 Manifest Class Header xxviii
Appendix 17 World Renderer Draw Method Implementation xxix
Appendix 18 Example C++ Unit Test and Visual Studio Result xxx
Appendix 19 Performance Analysis on the Server Project xxxi
Appendix 20 Performance Analysis on the Client Project xxxii
Appendix 21 Further Performance Investigation into Rendering Code xxxiii
Appendix 22 WireShark Packet Trace Result xxxiv

iv. Glossary of Terms

CLR	*Common Language Runtime – a component of Microsoft's .NET Framework responsible for executing .NET code.*
Dynamic Linking	*Used to describe how a component is used by application binaries - in this case, the component is included as a Dynamic Link Library (.dll file on Windows, .so on Linux and .dylib on Mac), and is loaded from disk at run time.* *See also: Static Linking.*
FTP	*File Transfer Protocol – a protocol describing the transfer of files using TCP/IP.* *See also: TCP, IP*
Garbage Collector	*An element of a language runtime responsible for freeing memory no longer in use.*
Header	*In C and C++, a header file describes methods and variables (and in C++'s case, classes and members) available to other code to use. Usually uses the .h extension, or in C++ .hpp.*
HTTP	*Hyper-text Transfer Protocol - a protocol describing the transmission of hyper-text documents on the Internet.*
IP	*Internet Protocol – describes a method for communication on the Internet.* *See also: TCP, UDP*
IPv4	*Version 4 of the Internet Protocol.* *See also: IP*
Lag	*In computer games, "lag" is typically used to describe network latency or delay, but can also be used to describe a low frame rate.*
libpng	*The PNG reference library.* *See also: PNG*
libpng Licence	*License covering the libpng library. Often referred to alongside the zlib license, since the two licences share commonalities.* *See also: libpng, zlib licence*
NAT	*Network address translation – A method of utilising remaining IPv4 address space by allowing multiple hosts to access the Internet via a single public-facing IP address.* *See also: IPv4*
NDA	*Non-disclosure agreement, also referred to as confidentiality agreement, or proprietary information agreement. Legal contract that binds two parties in an agreement to restrict access to a resource from third parties.*
PNG	*Portable Network Graphics – an image compression format licensed under the libpng license.* *See also: libpng, libpng licence*
Static Linking	*Used to describe how a component is used by application binaries - in this case, the component is included inside the application binaries themselves.*

	See also: Dynamic Linking.
TCP	*Transmission control protocol – connection-oriented, reliable delivery of packets on a network.* *See also: UDP, IP*
UDP	*User datagram protocol – connectionless, unreliable delivery of packets on a network.* *See also: TCP, IP*
Vertex *(pl: vertices)*	*Term used in Computer Graphics to define a point that describes corners of a 2D or 3D shape.*
zlib	*A library used for data compression.* *See also: zlib license*
zlib Licence	*A software license developed for zlib that is often used for other software libraries because of it being a free software licence.* *See also: zlib, libpng licence*

1 Introduction

1.1 Preamble

Multiplayer games are ever more prevalent in the current video game landscape, driven by the sales of big franchises such as Battlefield (Electronic Arts 2011) and Call of Duty (Activision 2013). Much focus is given to games containing multiplayer features, and profits from these products show that they are an important area of consumer investment.

The two main protocols on the Internet that facilitate inter-network communication are the transmission control protocol (TCP) and the user datagram protocol (UDP). The two protocols serve at the core of the Internet, facilitating the transmission of all types of data from everywhere in the world. (Tanenbaum 2011)

This project aims to cover the design, development and testing of a prototype game engine, with a full network implementation in TCP and UDP. The merits of each protocol will be examined, and analysis and benchmarking will be offered, providing a conclusion about which worked best for the prototype, and what further research needs to be performed in order to gain a better insight into the problem.

1.2 Aim

The aim of the project, as phrased in the project specification included as Appendix 1 states:

> *The deliverable for this project will be a prototype multiplayer networked game engine that uses modern networking techniques.*

Objectives describing what is required to meet this aim are offered in section 1.3.

1.3 Objectives

The requirements of this project pertain to the network protocols, however also include the implementation of other features in the prototype. This contributes to making the prototype as realistic as possible, for the purposes of testing.

Metrics that will be used to analyse how well each objective should be met are included below each listed objective, and will be referred to in the evaluation to demonstrate how well objectives were achieved. This serves as a reference point to analyse the quality and usefulness of the application.

1.3.1 Research and Compare Existing Software and Development Methods

One objective of the project is to research existing networking, software and development methods, comparing each, evaluating drawbacks and merits.

The metrics for measuring the completeness of this objective can be seen in the research performed in this document.

1.3.2 Compare Each Method and Tool, Evaluating Drawbacks and Merits

The research performed in section 1.3.1 will then be used to compare development methods and tools, evaluating the merits and drawbacks of each tool with the view of making an informed decision about which is suitable to use for the prototype.

Metrics for this objective can be measured as the inclusion of an extensive analysis and comparison of each method and tool described.

1.3.3 Decide Which Tool to Use, and How to Use it

The final tools will be chosen based on the comparison and evaluation in section 1.3.2. This stage will decide which tool is used in the development stages of the project, and is crucial to the prototype's design.

The metric for measuring the success of this objective in the final project can be observed as a final decision about each tool and technology, based on existing comparisons and research.

1.3.4 Conduct Interviews With Knowledgeable People

Interviews with knowledgeable people will be conducted as another objective for the project – gaining primary forms of research to design of the prototype is of paramount importance. The results of the primary research can be seen in prototype, as they will shape how it behaves, and its core design.

Metrics for this objective include the conduction of interviews with selected knowledgeable candidates.

1.3.5 Create an Engine Design Based on Existing Technologies

Using the above research, an engine design will be constructed to base the implementation from. The design will include class diagrams, basic code structure, and notes about how different sections of the prototype tie together.

The metric for measuring the completeness of this objective can be seen in the final prototype design, which will be based on existing research and technologies.

1.3.6 Implement, Test and Document a Multiplayer Game Engine Concept

The application will be implemented using the prototype design created in previous sections. The measurable output of this stage will be a functioning application prototype, implementing the following:

- A basic core engine design, implementing basic features of a game engine such as levels, players, physics

- A functioning network stack including a TCP and UDP implementation for the purposes of testing
- A basic renderer to draw game objects on-screen

The above goals are attainable using the tools, technologies and techniques learnt and researched in previous sections.

The metric that will be used for measuring the achievement of this objective will be to see if a multiplayer game engine prototype is created, including appropriate implementation, testing and documentation.

1.3.7 Consider Different User Limitations Throughout, Such as Packet Loss and Latency

In order to keep the project as realistic and as useful as possible, different limitations on network capacity will be considered throughout the prototype's design process. The measurable output of this will be demonstrated in the final product, and will allow it to operate within slow network connections, and connections where packets are frequently lost.

The metric for analysing the completeness of this objective can be seen throughout the report, and finally in the implementation of the prototype, which should cater for packet loss and latency.

1.3.8 Evaluate Quality and Usefulness of the Application

The application will be evaluated based on its initial objectives and aims. These will be measured by referring to their implementation and design and research based around the area. The objective will offer a critical view of the successes and failures of the project, and offer explanations and points to ensure this doesn't happen again.

1.3.9 Conclusions, Recommend Further Work

A conclusion will be offered about the application, and any work that needs to be undertaken in the future to compliment or further the research outlined in the project will be explored. This section will offer not just a reflection, but a look at work going forward, made possible by the work already performed.

1.3.10 Critically Evaluate the Project

A critical evaluation of the entire project will be offered, analysing the successes and failures of the project as a whole. This section will include all phases of research, design and implementation.

1.4 Task Plan to Complete Objectives

The timeline for the above objectives is outlined in Appendix 2. It includes a sensible and realistic view of when tasks should be started and completed, and displays how

long should be spent on each task. It also is a good indication about what order tasks need to be completed, as it demonstrates which tasks are dependent on others.

2 Tools, Technologies and Methodologies

Selecting the correct language to develop with is a crucial aspect of this project – it defines which platforms the code will run on, and it shapes the entire design process.

Below languages, tools and libraries that could be used to construct the prototype will be discussed.

2.1 Programming Languages

Selecting the appropriate programming language at the start of the project is important, as switching to another at a later date will be a lot of work, and will cost precious development time.

The below section illustrates some of the most popular languages in use for Desktop applications, and their merits and drawbacks.

2.1.1 C♯

C♯ is a proprietary, object-oriented language provided by Microsoft with its .NET suite. C♯ uses a JIT compiler at run-time, and includes garbage collection. (ECMA International 2006)

It is primarily designed to work on Microsoft Windows, Xbox 360 and Windows Phone, and in fact its most popular compiler and runtimes only support those platforms. (Microsoft Corporation 2013c) In order to use C♯ on Unix platforms such as Linux and Mac, third party tools such as the Mono Framework must be used (Xamarin 2013). However, these tools are a lot slower than their official counterparts in many areas. (Mihailescu 2010)

The C♯ CLR (Common Language Runtime) is not ideal for games programming as it includes a Garbage Collector: “The garbage collector underlying C# might work by moving objects around in memory, but this motion is invisible to most C# developers.”. (ECMA International 2006, p. 26) This does mean that while the garbage collection process is in operation, the application’s execution is frozen. (Microsoft Corporation 2013b) This makes it somewhat unsuitable for games programming, as an arbitrary freeze in execution will negatively impact the game’s frame and simulation rate.

It is possible to bypass Garbage Collection using “unsafe” code akin to writing C code. (ECMA International 2006, p. 425) However, this is akin to writing code in C or C++: “In a sense, writing unsafe code is much like writing C code within a C# program” . (ECMA International 2006, p. 425)

Microsoft Visual Studio is the official method of writing C♯ code, however it is possible to write code outside of Visual Studio and use Microsoft’s command line build tools to build the code. (Microsoft Corporation 2013a)

Personal experience with this language includes 1 years' experience acquired at a software company. That means in this instance there is nothing to re-learn, and while using it there is less scope for being stuck.

2.1.2 C++

Modern games are mostly written in C++: "industrial-strength 3D game engines are still written primarily in C or C++, and any serious game programmer needs to know C++." (Gregory 2009, p. 20).

C++ is an object oriented version of the language C, which was created in 1972. (Ritchie 1993) C++ differs from C because it allows for objects that have data fields and methods, whereas in contrast C only allows for procedural code. (Stroustrup 1997, p. 21)

Compilers for C++ are available for all platforms, and to make sure that the same code can be used on different platforms strong cross-platform compiler support is of paramount importance.

In contrast to the aforementioned C♯ language, C++ has native compilers for each platform – the proprietary Microsoft Visual C++ compiler for Windows and other Microsoft platforms such as Xbox, and GCC for Unix-based platforms including Mac and Linux. (Stroustrup 2011)

There are a plethora of proprietary and open-source IDEs for programming with C++, including Microsoft's Visual Studio, Xcode, Eclipse CDT, Code::Blocks and many more. (Stroustrup 2013) C++'s inception was in 1983, and because of its age and ubiquity, it has well-established support among different platforms and vendors. (Stroustrup 2013)

C++ does not have any form of garbage collection built in, and instead relies on the programmer to allocate and free memory. (Stroustrup 1997, p. 567) This means that any memory that is not freed after use will cause a memory leak in the program (where memory is still allocated, but the reference to it has been lost). (Stroustrup 1997, p. 247) However, without a garbage collector, the programmer can choose when to free or allocate memory, meaning that there are no arbitrary freezes caused by a Garbage Collection feature (as in C♯). This aspect of manual memory management is crucial in relation to games, as they need to render to the screen at a constant frame rate.

Personal experience with this language does not measure to C♯, however the experience can still be considered good enough to construct a prototype in C++.

2.1.3 Python with CPython

Python is an open-source programming language provided by the Python Software Foundation. (Python Software Foundation 2013)

Its main (and official) implementation is CPython, written by the same foundation. This implementation is to be used if recent language features are needed, as this implementation generally receives them first. (Rossum 2013, p. 3)

CPython offers compilation of Python programs into intermediate byte code, which is then executed on the CPython Virtual Machine. Code can either be distributed in intermediate byte code form, or in plain text. (Rossum 2013, p. 82)

There are Python virtual machines available for Mac, Linux and Windows, and from its conception CPython was designed to be cross-platform. (Willison 2007)

Python is a garbage collected language, and has no facility for de-allocating objects. Instead they are cleaned up by the runtime, in a similar fashion to C♯. (Rossum 2013, p. 15)

Personal experience with Python extends to small projects over a couple of years. During this time, it was found that the tools for debugging it are not as fully-featured as for C++, C♯ and Java.

2.1.4 Java

Java is an object-oriented, closed source programming language designed by Oracle Corporation. Oracle also have an official Java Virtual Machine, which executes Java byte code (compiled from Java code). (Lindholm et al. 2011, p. 2)

Other implementations are available, however the official implementation of Java is the Oracle Java Virtual Machine. Similar to CPython, this implementation generally receives updates first. It is supported across all major platforms. (Lindholm et al. 2011, p. 5)

The Java Virtual Machine includes a garbage collector, similar to Python and C♯. (Lindholm et al. 2011, p. 13) As discussed above, this may make it a bad choice for a game engine because of jitter introduced by garbage collection cycles.

Personal experience with Java hints that it is very syntactically similar to C♯, experience with one language complements both.

2.1.5 Language Comparison

To decide which language is best for the project, the criteria listed in the headings below will be used.

2.1.5.1 Mandatory Attributes

Each programming language *must* have the following features:

- Cross platform support (either with its compiler or runtimes depending on language)
- Support for object orientation; i.e. objects with methods and attributes
- A wide selection of libraries, both official and developer-contributed

- A complementing IDE that supports breakpoints and viewing the values of variables while the code is running to aid debugging

2.1.5.2 Desirable Attributes

Desirable attributes cover attributes that aren't absolutely necessary, however are likely to increase productivity.

- Small or no pre-requisites (client frameworks, redistributables, runtimes)
- Personal prior experience with the language
- Compile to native code

2.1.5.3 Useful Attributes

Useful features are features that aren't particularly needed, however will likely aid development.

- Wrappers for complex operating system features (threading, sockets)

2.1.5.4 Incidental Attributes

Features that are entirely surplus, and will likely not be needed.

- Support for creating installers from the language framework

2.1.5.5 Undesirable Attributes

This covers features that will definitely not be needed during development, and could even hinder development or slow down the application.

- Garbage collection
- Dynamic typing of objects

2.1.5.6 Language Functional Analysis Score Table

The below table shows scoring for the languages and categories discussed.

Table 1 - Functional analysis for programming languages

	Mandatory Out of 4	*Desirable Out of 3*	*Useful Out of 1*	*Incidental Out of 1*	*Undesirable Out of 2*	*Total Out of 11*
C♯	4*	1	1	1	-1	6
C++	4	3	0†	0	0	7
Python	3‡	1	1	0	-2	3
Java	4	1	1	0	-1	5

** Cross platform support is provided by Mono, an unofficial implementation of .NET* (Xamarin 2013)
† Libraries can be used to wrap this functionality
‡ IDEs are available, but aren't as fully-featured as for C♯ and C++, from personal experience

2.1.6 Language Selection

Based on the above comparison, it appears that C++ is the most suitable language, followed by C#, followed by Java and Python.

C++ is therefore the most appropriate language of choice for the prototype, and will be used to construct it. The decision has a direct connection with the next subject, libraries, and each library discussed will be chosen because of its compatibility with C++.

2.2 Graphics, Input and Sound Libraries

When writing software, it is not necessary (and perhaps ill-advised) to write every single component from scratch. Libraries are either pre-compiled or portions of source code that have already been written, and can be used in the application to provide a new service or to make existing aspects of construction easier and less error-prone.

Frameworks will be reviewed that could potentially aid the construction of the prototype, and the merits and drawbacks of each library will be discussed in a similar manner to programming languages.

Since some aspects of the project require writing code that can be very low level, pre-written libraries can be used to wrap that functionality up.

2.2.1 Graphics Technologies

With regards to graphics, there are two main technologies supported by all GPU manufacturers that can be used to construct the prototype. These are OpenGL, and DirectX.

OpenGL is an open source graphics specification written and maintained by the Khronos Group, Inc. (Segal & Akeley 2013, p. 2) It is a surprisingly unpopular choice for modern game development, despite it being cross-platform, and widely supported by most GPU manufacturers.

The opposing technology for graphics development in modern game engines is DirectX. It is a specification written and maintained by Microsoft Corporation, and only works on Microsoft Windows, necessitating a separate graphics pipeline for Mac and Linux, using OpenGL.

Since many libraries already exist to wrap up OpenGL and DirectX functionality, low level graphics code won't be written. It would be much more time consuming and error-prone than simply using a library instead.

As for which graphics technology to use, an engineer from Valve Software (a large US based video games creator and publisher) who recently ported a 3D engine from DirectX to OpenGL recommended that it is the best choice to use OpenGL from the start of a project. (Valve Software 2013).

A selection of libraries will be considered that lean towards Open Source, as that means if there are problems, the underlying code can be looked at to aid debugging (and potentially implement fixes).

2.2.1.1 Sample Code

To test the libraries, each library will be tasked with drawing an 800x600 pixel window containing a green 400x200 rectangle in the top left (reference rendering included as Appendix 3). This will test the library and its ability to construct a 2D image on-screen.

This task will test how simple the library is to work with. It will show how easy the code is to read, and how much code is required to perform a very simple task.

As a benchmark, an example has been constructed using pure OpenGL code, included in Appendix 4.

The code is very verbose, and difficult to read. The rectangle is constructed using its component vertices, and a lot of boilerplate code has to be used to construct, create and clear the drawing canvas. The code is a total of 43 lines.

In order to construct the prototype, a library will be used to simplify writing the graphics code. The library will satisfy the following goals:

- It must be readable, hence easily maintainable
- It must be as expressive as possible, hence as few lines as possible
- It must be as simple as possible

2.2.2 SFML

Simple Fast Multimedia Library (SFML) is an Open Source library that provides a simple interface for components on a PC.

It runs on Windows, Linux and Mac, and uses OpenGL behind the scenes. Written in C++, it has official bindings for C, C++ and .NET (including C♯), and has community contributed bindings for Java, Ruby and Go. (Gomila 2013b)

Figure A exemplifies a very simple usage of SFML (Simple Fast Multimedia Library) in C++, to create the 800x600 pixel window and draw a green 400x200 pixel rectangle onto it.

2.2.2.1 Licensing

SFML (Simple Fast Multimedia Library) operates under the zlib/png licence, meaning that it can be used for any means without any restriction. If the project ever evolves into a commercial endeavour, this library could be used without issue. (Gomila 2013a)

2.2.2.2 Sample Code

In 23 lines of C++ code (included as Appendix 5), a window was created that listened to events and drew a shape on screen. This means the library's syntax is concise enough for use as a simple renderer in the prototype, and simple enough to understand at a glance.

Compared to OpenGL, the code above is over 20 lines shorter, and includes fewer method calls as a result of less boilerplate code. It is also far more expressive, as SFML includes classes to wrap up simple shape drawing such as the `RectangleShape` class.

In contrast, OpenGL has no awareness of shapes, and requires every drawn object to be constructed using vertices, in a matrix-like format.

2.2.3 Allegro

Allegro is a cross-platform, open source game programming library for C and C++ developers.

Like SFML, Allegro handles creating windows, user input, graphics and sound. It uses OpenGL over DirectX on Windows (similar to SFML). Its original developer was Shawn Hargreaves for the Atari ST platform, but open source contributions have continually improved the project. It is now a powerful 2D and 3D game library. (Harbour 2007, p. 36)

2.2.3.1 Licensing

Allegro v5 is licensed under the zlib license, similar to SFML. This means it can be used for any purpose, including personal and commercial endeavours making it a good choice when considering libraries for the prototype.

Similar to SFML, it uses OpenGL behind the scenes, and can be used to wrap up OpenGL calls using the library's friendlier methods.

2.2.3.2 Sample Code

In only 33 lines of C++ (included as Appendix 6), the reference experiment has been implemented in a similar manner to SFML, and low level OpenGL calls haven't been resorted to.

If the code is inspected however, in comparison to SFML, it is far less expressive. In SFML, a rectangle object is created, it is coloured, and then the window is told to draw it. With Allegro, the call to draw a rectangle is contained within one line, and has to be constructed each time the application enters its drawing loop.

This is very similar to OpenGL calls, as they are not object oriented, and instead are pure C functions that operate procedurally. The goal of using a framework is to allow the use of object-oriented programming when constructing the graphical side of the application.

To that end, this library is less suitable that SFML for the project, however is still a viable choice if pitfalls are encountered with SFML.

2.2.4 Qt

Qt is a cross-platform framework used to develop GUIs for applications and command line tools. It is available under GPL v3, LGPL v2, and a commercial license, and is developed by Digia and the Qt Project (a group comprising of individual developers and companies developing Qt).

Its commercial arm was originally owned by Nokia, who sold it to Digia in 2012. (Digia Plc 2012)

Qt allows developers to build interfaces using Qt widgets, and its SDK also includes an integrated IDE that simplifies the interface building process using a drag and drop-style mechanic.

As the prototype may at some point need a GUI, it could simplify that process, if Qt widgets and its IDE were used. The only other library mentioned that also provides this feature out of the box is Allegro, however Allegro doesn't include an IDE.

2.2.4.1 Licensing

Qt offers 3 licenses, each differing in the following ways:

- GPL v3 – All source code must be disclosed: nothing may remain private
- LGPL v2 – The Qt libraries must be dynamically linked if source code will remain private
- Commercial Licence – Qt can be used however the developer wishes. Source code may or may not be disclosed, the libraries may be dynamically or statically linked

When considering Qt, its license must also be considered. For instance, if it was decided that the prototype would be sold for commercial consumption, the Qt binaries must be distributed separately (as dynamic link libraries), or if they are to be used statically the source code of the prototype must be published in full.

Of course, at that point a commercial license may be considered, however it is not wise to tie down projects financially before they have launched – so in the case of Qt, a good compromise between cost and openness is to distribute the Qt binaries dynamically with the prototype.

2.2.4.2 Sample Code

The reference code is included as Appendix 7. On the face of it, this is by far the most concise code for drawing the rectangle. Similar to SFML, a `QRect` object can be observed, along with a `QColor` object to wrap up the colour drawn.

However, behind the scenes, this is actually more complicated than the other libraries tested. To get to that stage, the project consists of 4 different files:

- `mainwindow.h` – The header for the code representation of the main window
- `paintwidget.h` – The file shown in Appendix 7, the `PaintWidget`
- `main.cpp` – The `main()` function
- `mainwindow.cpp` – The code for `mainwindow.h`
- `mainwindow.ui` – The code generated by the QT UI designer, describing which widgets are on-screen (in this case, `MainWindow` with the child `PaintWidget`)

This is quite a complicated project structure compared to other libraries, as the project has been forcefully architected around a user interface. That could be quite a drawback when writing game code, as it will be architected very differently from UI code.

2.2.5 Library Comparison

To decide which library to use, the below criteria will be used, similar to the comparison of programming languages:

2.2.5.1 Mandatory Attributes

- Cross-platform support
- Be object oriented (not pure C)
- Not overly verbose
- Clear to read

2.2.5.2 Desirable Attributes

- Networking, audio and graphics support
- Completely open source (code is freely available to view, no distribution restrictions

2.2.5.3 Useful Attributes

- A built-in UI library

2.2.5.4 Incidental Attributes

- A specially designed IDE

2.2.5.5 Undesirable Attributes

- Mandate a specific code design and class layout

2.2.6 Library Functional Analysis Score Table

The below table shows scoring for the languages and categories discussed.

Table 2 - Functional analysis score table for libraries

	Mandatory Out of 4	*Desirable Out of 2*	*Useful Out of 1*	*Incidental Out of 1*	*Undesirable Out of 1*	*Total Out of 7*
OpenGL	1	1	0	0	0	2
SFML	4	2	0	0	0	6
Allegro	4	1	0	0	0	5
Qt	4	0	1	1	-1	5

Based on the above evidence and the functional analysis score table, SFML is the clear choice to use for a library.

2.3 Communication Protocols

Communication protocols will form the basis of the network layer of the prototype. The use of libraries will abstract the usage of protocols in the application, however an understanding of the fundamental inner-workings of each protocol is still required to ensure they are not misused.

2.3.1 TCP/IP

Transmission control protocol (TCP) is the basis of the Internet. It is used for reliable communication between hosts, as it ensures sequential, guaranteed delivery of packets using the concept of a "connection".

TCP (transmission control protocol) is used for Hyper Text Transfer Protocol (HTTP) traffic on the Internet, and is suitable for applications where packets cannot be dropped or lost. The added reliability adds some overhead to the protocol however, as packets are transparently resent if lost and buffered if received out of order. (Comer 2005)

Its overhead makes it unsuitable for applications where real-time data transmission is required over reliability – for instance, voice over the Internet Protocol (IP), or first-person shooter games.

2.3.2 UDP/IP

User datagram protocol offers unreliable, connectionless communication between hosts. Packets are not guaranteed to arrive at the destination in order, and in fact aren't guaranteed to arrive at all. (Tanenbaum 2011)

UDP is used for situations where data is required in a time-sensitive fashion, where reliability does not need to be ensured. Examples of such situations are media streaming such as voice over the IP (Internet Protocol), and games where actions need to be processed within milliseconds of them occurring on a host (first-person shooters) in order for the game to remain fair.

It is unsuitable for such applications as Email, HTTP (Hyper-text Transfer Protocol) and file transfers via File Transfer Protocol (FTP).

2.3.3 DCCP/IP

DCCP is a version of UDP, that allows for congestion control in addition to fast and unreliable packet transmission. It is designed for applications where delay is not desirable, such as online gaming and streaming media (similar to UDP). The protocol is different because it includes a flow-control like feature, similar to TCP. Flow control allows the protocol to adaptively change the flow of data, so end-devices unable to receive at the transmission speed of the sender do not become overwhelmed with data. (Kohler, Handley & Floyd 2006)

DCCP is a relatively new protocol, and finding libraries that offer support for it is difficult. It also lacks support and documentation.

2.3.4 Protocol Comparison

While DCCP does offer some desirable features, its lack of support in both libraries and the community is a large factor to consider, as it affects how much information and help is available for implementation. Therefore it is unwise to implement this protocol into the prototype.

Since the two main protocols in this area both offer desirable and undesirable features in equal measure, the engine will implement both protocols.

This is because games created using the prototype engine may require either protocol – for instance, if a chess game is built, TCP would be desirable for its reliable properties. However, if a shooting game is built, UDP would be desirable for its speed.

Implementations of both protocols will be offered transparently to the rest of the engine, and such a switch will be easy to perform, depending on the client's requirements.

2.4 Physics Libraries

When designing a 2D game, it is important to consider different aspects of its gameplay. Something that is common to most modern games (2D or otherwise) is the idea of physically simulated objects, which react to forces such as gravity and other objects in a realistic manner.

To achieve this effect, it is necessary to implement a physics simulation layer into the game engine. However, achieving a bug-free realistic simulation is no easy task, as the mathematics involved in such an endeavour require a lot of research and testing. Therefore, in a similar vein to the Graphics, Input and Sound libraries, an existing physics simulation library will be selected for use within the engine.

Using the library, it will be possible to create interesting level designs using simulated objects, forces and constraints. For example, it would be possible to have boxes that the player can knock down, or a seesaw object that can be jumped on.

2.4.1 Havok Physics

Havok Physics is a 3D physics engine written and distributed by Havok, which is owned by Intel Corporation. (Intel Corporation 2007) Havok Physics offers robust and fully-featured collision detection, with support for large collision meshes and spatial queries. (Havok 2013a)

It is widely used in the gaming industry, and has been used in popular games such as Saints Row IV, Assassin's Creed III, Call of Duty: Black Ops II, and games built using the Source Engine (a product discussed in section 2.5.1). (Havok 2013b)

Havok is a commercial physics engine, and the details of its price are subject to an non-disclosure agreement (NDA). Not knowing the pricing of the physics engine it makes it a very unattractive library to consider for the prototype.

2.4.2 Box2D

Box2D is a freely available 2D physics and collision detection library developed by Erin Catto. It is licensed using the zlib license, and features continuous collision detection, convex shape support, friction, joints, momentum, reaction forces and more. (Catto 2013)

Box2D is written using portable C++ and can be built as static or dynamic libraries, and it does not depend on any particular standard library. Given the scope of the project, Box2D is a better choice than the commercial Havok Physics, as it does not incur a license fee of an undisclosed amount.

However, being a free library, it means that there is not a direct line of support with the library, as would be provided with Havok. Box2D is used in many commercial and free projects, including the popular mobile app "Angry Birds", (Kumparak 2011) meaning that there is lots of community-driven support available.

If a library such as Box2D is used, it will necessitate the writing of wrapper functionality, if the engine is to be capable of both 2D and 3D simulation in the future. This way, the implementation of Box2D can be swapped with a 3D physics library such as Havok, with minimal code changes in other parts of the application.

2.4.3 Physics Library Comparison

Due to the commercial nature of Havok, and the nature of the prototype, Havok is unsuitable for use in this project. It offers a complete physics implementation, but at an undisclosed cost.

Box2D is the library that will be used in the prototype – it is open source, and doesn't impose any requirements on applications using it.

2.5 Similar Works

In order to construct a prototype game engine, it is wise to first analyse other offerings in the same domain. This allows an insight into how other successful game engines are architected, and what kind of features they have. The following section serves as a discussion to that end, designed to simply analyse similar offerings, and offer suggestions as to which bits of engine architecture can be built upon and used in the prototype.

2.5.1 Source Engine

The Source Engine is a 3D game engine from Valve Software. It has been used for games such as Half-Life, Portal, Counter-Strike and Left 4 Dead. The engine was created for Half-Life 2 in 2004, and since then has been receiving regular updates to improve its feature set.

It is a vastly bigger project than the prototype, however was selected for analysis because of personal experience with the engine. This experience with Source spans over 2 years, and includes a successful public game release built with the engine (Estranged: Act I).

The public-access portion of its source code is a very useful insight into how games are architected under the hood, so to speak. Since the engine is versatile enough to have been used on several commercially-successful titles, there may be something to learn from the architecture of such a piece of software.

2.5.1.1 Workflow

With this particular engine, all level editing is performed using an external level editor, that then compiles levels to a format that the engine can understand once the level designer has finished.

For the prototype, although a level designer would be a useful feature, it may be overkill for such a small project. In the case of the prototype, it is a 2D engine, so a level editor may not be absolutely necessary anyway, as the cognitive load involved with conceptualising objects in a 2D space is far less than in 3D.

However, ultimately a level editor is a very important aspect of a game engine, and if the prototype is to eventually be sold, it will be a very appealing feature to developers selecting an engine – so should be a feature that is seriously considered.

2.5.1.2 Code Layout

The most useful aspect of this engine is its code structure, as this is the part that will aid the most in designing and building a prototype.

The engine's source code is available in part via a public download, with the goal of providing developers a platform for which to develop small games or "mods" for Valve's games, such as Half-Life: 2.

Looking at the publicly available C++ code for this particular engine, it can be seen that it uses an inheritance structure centred around entities, with a client/server architecture.

The client/server architecture allows for authoritative logic on the server, and predictive logic on the client to counter the effects of latency (this topic will be discussed further in future chapters).

The class diagram included as Appendix 8 demonstrates the different classes and interfaces involved in a simple physically simulated object in the Source Engine. This could be for instance a bucket that responds to gravity, the player, and other physically simulated entities.

The diagram shows classes from two projects; the client and the server. The code for both libraries is run separately. In the case of single player games based on the Source Engine, a server and client are run locally (with no prediction since locally there is no latency). In the case of a multiplayer game, the client code is run on a player's computer, and the server code is run elsewhere, typically on dedicated server hardware.

As seen in the diagram, there are lots of layers of different classes due to the features required by the base classes. For instance, at the bottom of the chain, the first port of call is the `CPhysicsProp`. It, in turn, may represent a breakable prop, such as wood boards that the player can smash, so inherits from `CBreakableProp`. That inherits from `CBaseProp`, and that from `CBaseAnimating` (the class responsible for all game objects requiring a model). The next item in the chain is `CBaseEntity`, followed by `IServerEntity`, `IServerUnknown` and `IHandleEntity`.

Within the client code, the class hierarchy is similar, but is instead driven more towards rendering. (It should be noted that the convention in this particular engine is to name server code `CClassName`, and client code `C_ClassName`.) For instance, `IClientEntity` implements the `IClientRenderable` interface, which the server has no need for.

The scope of the Source Engine has a bearing on its class hierarchy, and the prototype game engine will likely not need as much code, or layers of abstraction. However, it is still a viable template to use for designing a prototype engine, since the engine has been used to construct so many varying games.

2.5.1.3 Networking

The networking aspect of this engine is also of paramount importance, as it is one of its major features, and something that can be used to help with the construction of prototype.

As seen in the class diagram, classes in the Source Engine have a server and client counterpart. This allows for separate code to be logically related – so for instance a

dynamic light entity would be represented as `CDynamicLight` on the server, and `C_DynamicLight` on the client.

The server class can be seen in Appendix 9. It is fairly simple, and includes some light variables, whether the light is on, and certain options for it. It also inherits from `CBaseEntity`, meaning that all of the networked values from that class are also sent along the wire to the client. These include global variables such as position, velocity, angles, etc.

Such data is needed by most entities in the game, so it makes sense to only define it in the base entity class.

The corresponding client entity, shown in Appendix 10, contains the actual dynamic light (`dlight_t`), and merely acts as a receiver of the variables sent from the server, for use in the rendering pipeline.

This architecture is similar from the engine that Source is derived from, which is the engine used in the video game Doom. It is a model that describes an authoritative server, responsible for simulation, and a client responsible for rendering (and latency compensation). (Waveren 2006)

2.5.2 Torque2D

Torque2D is an Open Source 2D engine that allows for cross-platform game development. It includes facilities for physics, sound, graphics and networking. This section will look at the typical workflow for creating a scene using Torque2D, and how it differs from previously discussed offerings. (GarageGames, LLC 2013)

2.5.2.1 Workflow

Torque2d includes a level editor, referred to as "Torque Game Builder". Similar to Source's Hammer level editor, it allows for objects to be placed in the world using an easy to use graphical interface. However, this is a feature of the paid-for pro version of Torque2d, which is an aspect that needs to be considered when assessing the appropriateness of this engine.

Torque2D has its own scripting language called *Torquescript*. It is very similar to C++, and in fact supports many of the features of C++ such full object-oriented programming support and mathematical functions.

2.5.2.2 Code Layout

As mentioned above, the preferred method to write code for this engine is using *Torquescript*. This is very different from the previously considered engine, since the Source Engine uses code that is compiled before it is distributed. Torquescript is instead interpreted by the engine at runtime, meaning that the source script code is distributed with the game.

2.6 Development Methodologies

When designing software, there are several popular development methodologies to pick from. When creating the prototype, it is important to consider different methodologies to increase productivity, and to ensure that the prototype's development is kept on track.

2.6.1 Scrum

The Scrum methodology was first published by Ken Schwaber and Jeff Sutherland in 1995. It describes a "lightweight", "simple to understand" and "difficult to master" process framework that describes a set of rules to abide by during the development of products.

It describes events for "inspection and adaptation" contained within a "Sprint". These are:

- Sprint planning – Create a sprint goal.
- Daily scrum – 15 minutes time boxed event for the development team
- Sprint review – Held at the end of the sprint
- Sprint retrospective – Scrum team to create a plan for improvement

Each sprint should last no longer than one month, and during the sprint, no changes are made that endanger the goal.

(Sutherland & Schwaber 2013)

2.6.2 Agile

Agile is a methodology created by a range of authors. It has been traced back to 1957 at IBM's Service Bureau Corporation. (Larman & Basili 2003)

Agile is a general methodology, but relates to software development in the following ways:

- Individuals and interactions are preferred over processes and tools
- Focus on working software instead of comprehensive documentation
- Customer collaboration instead of contract negotiation
- Don't follow a plan – respond to change

(Beck et al. 2001)

These are the values that a project conforming to the Agile methodology should take into account and implement.

In terms of developing a prototype, it can be taken to mean the following:

- Requirements change, even in late development
- Primarily, the measure of progress is working software
- Simplicity is essential

- Business and development must collaborate
- Face-to-face communication is the easiest form
- Pay attention to good design
- Working software should be frequently delivered

(Ambler 2014)

2.6.3 Rapid Application Development

Rapid Application Development (RAD) is a methodology that describes designing and developing an application that meets requirements within a short period of time. The application evolves throughout development, typically based on continual customer feedback across the entire development process. (Gottesdiener 1995)

It differs from other methodologies as work is delivered as a chunk at the end of the development cycle. This is a change from other methods, where work is delivered in milestone chunks – as with Scrum (section 2.6.1).

2.6.4 Methodology Comparison

Three methodologies have been explored – Scrum, Agile and RAD (Rapid Application Development). All include elements designed for working in a team – however, RAD seems best suited to the prototype because it allows for fast iteration, and constantly evolving requirements.

Therefore, this is the development methodology that will be used to develop the prototype.

2.7 Impact on Design

The research performed in this chapter will provide a basis for the initial design of the prototype. All selected technologies in this chapter will be used, after undergoing careful research and investigation.

Any changes at a later date will be difficult, as large portions of code may have to be re-written. However, additional libraries and technologies may need to be used – they will be discussed in the design and development sections of this document.

3 Research Approach

The research into languages, libraries and existing technology is insufficient to begin constructing a prototype. To obtain more insight into such an endeavour, it is necessary to consult other mediums to gain a far broader idea of the item being created.

For that, research subjects must be acquired, and simple meetings must be set up and structured accordingly with each subject.

For the prototype, it will be important to construct interviews with relevant faculty members of Sheffield Hallam University who have expertise in similar areas. It is also intended to speak to members of staff from previous experience at a software house, as some have the relevant experience with game development.

3.1 Forms of Research

The type of research that is needed for this project is broad, but is designed to achieve one end goal: to inform the design and development of the prototype. This stage of research will draw upon the experience and knowledge of subjects that have intricate knowledge of the subject area, and will be able to reliably inform its design. The form of research that best informs this process will be selected.

3.1.1 Inductive or Deductive Reasoning

Two types of reasoning are described for consideration when performing research:

- *Inductive* research is the act of completely deriving an idea or hypothesis from data; the research is not informed by any other prior research performed.
- *Deductive* research describes the research method in which an hypothesis is derived from existing theory; this can in turn inform the data collection process.

(Bryman 2001, p. 8)

Inductive or deductive research could be used to complete the research for this particular project, however deductive reasoning is the most suitable. This is because in this area, software development, a lot of research and documentation already exists around the same problems, the research that is performed here compliments it.

It would be unnecessary to disregard all conclusions garnered by other sources and solely trust the conclusions from the research below, and would also leave many places where research had not been performed. Therefore, deductive reasoning will be used.

3.2 Data Collection Methods

3.2.1 Questionnaires

A questionnaire is designed to measure; it is a tool to collect data. Questionnaires are generally distributed to participants, and participants are left to complete and return the completed item.

Advantages of a questionnaire include low cost, easy processing, less interviewer bias, and a potentially large reach. Drawbacks include the fact that there is no way to correct misunderstanding, no control over the ordering of responses, incomplete responses, or unintended spread of the questionnaire material.

In order to design a good questionnaire, questions must be constructed in such a way that does not lead candidates, and the selection of candidates has to be appropriate and representative.

3.2.2 Interviews

While surveys are important for collecting information, it is also necessary to interview candidates. The main advantage of using an interview rather than a survey is that open-ended questions can be asked. Surveys are good for acquiring closed answers to set questions, interviews are good for gaining a broader insight into a chosen topic. (Oppenheim 2001, p. 81)

Below is a discussion of each interview type, and its appropriateness in relation to the research for the prototype's design.

3.2.2.1 Exploratory Interviews

Exploratory interviews – a less structured interview, this method is designed to develop hypothesis and aid idea development, rather than to collect data. (Oppenheim 2001, p. 67)

This method of research is a suitable approach when conducting fact-finding and acquiring design ideas for the prototype. Using it, questions can be vague and open in nature, allowing the interviewee to answer with any brevity or verbosity they desire.

This is very useful for the prototype, as it allows for the interviewee to cover topics other than the main subject of the question, so if something is mentioned that is relevant to an area of the prototype's development but not strictly part of the current discussion, it can be noted.

Exploratory interviews are inductive, as they are not based on any particular prior research, they are designed to inform future research. For this particular project, exploratory interviews may not be the most suitable research method. Instead it would be better to use a deductive research approach, as discussed in section 3.1.1.

3.2.2.2 Semi-Structured Interviews

Semi-structured interviews lie between exploratory interviews and fully structured interviews (discussed in section 3.2.2.3). This type of interview is the kind that will inform the project, as it allows for structured questions that invite open answers.

Semi-structured interviews give scope for "jog" questions, questions that are designed to acquire information out of interviewees if they haven't completely answered the question at hand. This is a very useful tool, and without it some interview responses may not be as useful as they could be, creating the need for more data collection.

Questionnaires (section 3.2.1) can also be used as a confirmation of findings in semi-structured interviews, as they can be sent to other participants to attempt to make findings align, and normalise hypothesis.

3.2.2.3 Fully-Structured Interviews

This type of interview can essentially be described as data collection. This type of interview is usually conducted with well tested questions to a select sample of people, each with the same questions (hence "standardised").

Questionnaires are similar to this type of research method, differing only in the situation in which they are performed (questionnaires are generally unassisted).

It is less suitable to inform the construction of the prototype, as the results of this process will be used to inform design and development decisions rather than inform an investigation. This type of questioning is better suited for analysis, and if used will likely garner artificially restricted data that isn't suitable for its purpose.

3.2.3 Participant Observation

Participant observation is a technique widely used in social science research. The technique describes an observer immersing themselves in a group of participants, listening to what is said and watching the actions of those taking part. (Bryman 2001, p. 292)

Two scopes of participant conversation can be used:

- *Covert* – Observation performed without the explicit knowledge of participants
- *Overt* – Observation performed with the knowledge of participants

Covert observation is unsuitable for this project, as it is only necessary where the overt observation of subjects may affect data collected from them. There are also ethical concerns with covert observation that would have to be addressed, such as lack of consent.

Participant observation is not suitable for the development of the engine itself; however is a suitable mechanism for any games that are created using the engine. This is a very important aspect of game design, and play testing allows developers to observe which sections of the game work, and need changing.

Participant observation is therefore suitable for further work, however not for informing current engine development.

3.2.4 Case Study

A case study is an extensive analysis of a single "case". A case is commonly a location, and includes an extensive analysis of a setting. Case studies generally employ qualitative research methods rather than quantitative counterparts, such as participant observation (3.2.3) and unstructured interviews (3.2.2.1). (Bryman 2001, p. 49)

To that end, case studies employ most research techniques available, so all prior sections are applicable when utilising this research method. The implication of that is a case study is designed for collecting a very wide range of data with extensive analysis.

A case study is not appropriate for this type of project, as it denotes a very long term period of research into an established, observable topic. The research scope for this project is much smaller than an entire case study, so to pursue it would be unnecessary.

3.3 Evaluation of Data Collection Methods

Various data collection methods have been discussed in this section, including Interviews, Participant Observation and Case Studies. The suitability of each method depends on the type of project, and the required data for analysis.

For this project, a questionnaire provides too little intervention and means to provide responses. An interview is a much more appropriate technique, as candidates with relevant experience can be spoken to using a semi-structured interview, whereas candidates are presented with open questions, and invited to deviate.

3.4 Research Approach Choice

The most appropriate research approach in this case is dictated by the project type. For a prototype software development of this type, semi-structured interviews (section 3.2.2.2) are the best choice, as the method provides the best balance of structure and flexibility.

The same questions can be asked to each candidate, but the format allows for follow up questions and clarification as needed, which in this project will greatly aid the quality of data being returned.

3.5 Impact on the Project

The research collected constitutes raw data; the data collected from interviewees. This data will be processed, where it becomes information. The information is read and understood, and it becomes knowledge. The knowledge is then applied to the design of the application, and is then referred to as intelligence.

This cycle is important, as it shows the flow of information from beginning to end; research to application. This is how the research will be used an implemented in terms of this project.

4 Informing the Design

The design of the application will be informed by the research to be performed, using the selected research approach described in previous sections.

4.1 Areas that Require Research

Primary research is required in areas that personal knowledge is lacking, or areas that are not available to research. This could be for a number of reasons, however is commonly a problem in this area because of other works being closed-source, so they cannot be analysed to inform the prototype design.

4.1.1 Handling Dependencies

Dependencies are common in a game engine. Separate systems often work with each other, meaning that they need to know how to reach other services. This is an ever-evolving area in software development, and ventures past technical limitation to abstract software design.

With object oriented languages such as C++, problems with design are often much more apparent than problems with implementation. To that end, a question has been devised to pose to interview participants with relevant experience, to attempt to get a good answer to the problem;

> *When structuring a game engine, different components of that engine have to interact, and most components will have dependencies on other components. Which design pattern best suits the management of this?*

The question succinctly sums up the problem of how dependencies should be handled, and hints at the suggestion of using a pre-existing design pattern to solve the problem. This would require further reading, and would allow for additional sources to be consulted.

4.1.2 Language Decision

The language that is planned for use in the development of the project is C++, as it appeared the most suitable tool for the prototype. However, this could indeed be an incorrect decision, and the interview participants may be able to suggest a better alternative to research.

A question describing the problem of language selection appears as the below:

> *Do you think C++ is a good language decision for a prototype 2D engine with networking?*

The question does include a slightly leading suggestion that may skew results, however, since C++ appears to be the best tool for the project (based on the research in section 2.1.2), at this stage this decision needs to be confirmed.

As a semi-structured interview is the medium being used for research, if candidates state that they recommend alternatives to C++, a follow up question can be asked regarding which language they'd recommend, and why.

4.1.3 Network Packet Size

An area that is very difficult to acquire data for is specific technical challenges that may be faced, particularly regarding networking. This is because popular commercial game software is closed source; that is, generally their internal code is unavailable. Even when code portions are available, the network stack is not included (as with the Source Engine, see section 2.5.1).

To that end, a question has been constructed regarding a specific area where personal prior knowledge is lacking:

> *How large should individual network update packets be, and how frequently should they be exchanged?*

This question asks the participant to reflect on their experience, and offer an answer to the specific network problem of packet size, and packet frequency. This will inform the design of the application because it will hint at how many object updates will be sent in each packet, and how often the game's main loop should update connected clients.

4.1.4 Network Packet Frequency

This area is similar to section 4.1.3 because it is also a specific technical question requiring an answer derived from prior experience.

> *With regards to a "main loop", should threading be used to separate rendering and simulation? When should network packets be processed?*

This question attempts to answer the problem of threading in a game engine. Threading is a feature of operating systems that allows operations to run simultaneously. It is often difficult to implement as sharing data between threads has to be given special consideration, as shared resources have to be made "thread safe".

4.2 Interviewees

Interview subjects will remain anonymous throughout the interview process. The subjects will be selected based on their professional and personal experience in the area of game development. All candidates have worked on game projects in the past.

In order to properly inform candidates about the project, and acquire consent for their involvement, an interview background and consent form will precede all recorded interactions. The form (which requires a signature from participants) is included as Appendix 11.

4.3 Research Results

Appendix 12 contains the results of 5 different interviews with members of the industry with experience in the area of game design. They are colour coded according to their role to distinguish each participant, but still offer anonymity:

	Lecturer with prior experience working as a team lead for projects at Gremlin Games
	Software engineer with experience working for Disney games studio
	Software engineer and architect with experience at various software development houses
	Freelance 3D artist with experience with the Source Engine, Unreal Engine and CryEngine
	Freelance game developer working on 3D first person shooter Insurgency, and Estranged

The answers to each question will be analysed below.

4.3.1 Question 1 Analysis

Q1. When structuring a game engine, different components of that engine have to interact, and most components will have dependencies on other components. Which design pattern best suits the management of this?
This answer mentions "The most generic design I've seen is Unity which uses the entity-component model". On closer inspection, this answer recommends a pattern that prefers a composition code structure (flat object hierarchy implementing desired interfaces) over a hierarchical one (parent classes, children inheriting). This is a slightly different topic to what was asked, but is still a good suggestion. The participant suggests the following techniques: singletons, the observer pattern, and pure abstract classes. The participant also notes that global variables are "terrible" for managing dependency.
This answer recommends dependency injection, singletons, and service locators. The answer also notes that global variables are undesirable, and mentions the SOLID principal.
This answer mentions the service locator pattern, with each service implementing a specific interface. The answer also mentions leveraging idle time for simulations.
Dependency injection is described favourably in this answer, with the justification of easier debugging.
Global variables are mentioned in a negative light in this answer, however singletons and service locators aren't.

4.3.2 Question 2 Analysis

Q2. Do you think C++ is a good language decision for a prototype 2D engine with networking?
This participant notes that C++ is a good choice depending on the goals of the prototype, and the knowledge of the programmer.
This answer states that Java or Microsoft XNA would be better suited, as they preclude the need for boilerplate code.
This participant notes that Microsoft XNA may also be a good choice, however notes that they also used C++ in a similar project.

This participant notes that C++ would be their choice because of its speed, and discourages the use of Java and C#.
This interviewee notes that libraries will be crucial when constructing the prototype, and ease of prototyping is entirely dependent on the code structure.

4.3.3 Question 3 Analysis

Q3. How large should individual network update packets be, and how frequently should they be exchanged?
This answer states that the frequency of packets is dependent on the deployment of the game.
This participant states that it would be a good idea to make the update speed configurable.
The answer notes that the maximum transmission unit (MTU) should be taken into consideration. The interviewee also notes that the desired usage of the engine should be taken into account when setting the update speed / frequency.
This answer mentions that the MTU (maximum transmission unit) should be taken into consideration, and that this setting should be used when determining the size of packets. It is noted that another product, the Source Engine, sends packets 60 times per second.
This answer notes that 30 times per second is adequate, however more may be necessary.

4.3.4 Question 4 Analysis

Q4. With regards to a "main loop", should threading be used to separate rendering and simulation? When should network packets be processed?
Rendering and game logic should be in the same thread according to this participant, and a system allowing systems to run at different rates in the same thread is proposed. It is then said that network code should be executed in a separate thread.
“Heavy” tasks should be executed in a separate thread according to this answer in addition to networking.
Threads should be used for drawing, updates, networking and sound as a minimum according to this answer. The participant states that modern CPU’s have multiple cores, and they should be leveraged as necessary. An example of thread-safe code is given.
This answer states that the main loop should run updates, and separate threads should be used for physics and rendering.
An interesting point is raised in this response about the order that packets should be processed. It mentions that in order to use prediction, packets must be processed after game events. This must be considered if prediction is implemented into the engine.

4.3.5 Question 1 Conclusions

Composition should be explored over inheritance, and global variables are unanimously held in negative regard. Three of the answers given mention that service locators and singletons should be explored. However, the first participant mentions that singletons aren’t as good. Service locators will be a concept explored for dependency injection.

4.3.6 Question 2 Conclusions

Some participants suggest that C++ is a good choice. Two participants suggest it is dependent on the developer's experience, but overall the responses suggest that the language is the fastest and most efficient tool (in terms of processing and file size) for the job.

The design of the prototype will use C++, based on the primary research performed with the interview participants, and the research in section 2.1.2.

4.3.7 Question 3 Conclusions

Answers suggest that this is entirely dependent on the nature of the game, and one answer hints that it should be configurable. Therefore, it is wise to tweak the update send and receive rate until the game feels responsive.

For the prototype, it may be wise to start at 30 updates per second (as suggested by one participant), and change the value if necessary.

4.3.8 Question 4 Conclusions

All answers suggest that network operations should be performed in a separate thread, however answers do not offer a unanimous conclusion about which other systems should be threaded. One answer states that all "heavy" operations should be performed in a separate thread, and this fits with the other responses (some suggest rendering and physics should be performed in a separate thread).

For the prototype's design, it may be wise to begin with a single-threaded architecture, and move systems to separate threads as necessary.

4.4 Interview Conclusions

Overall, interviews were a success, and hinted at many different and so far unexplored aspects of the possible development approach. Interesting points were highlighted by more than one answer, and allowed for an informed consensus around the concepts of dependency management, language choice, update speed and threading.

The results have been concise yet very to the point, and will be used in the design phase of the prototype.

4.5 Impact on Design

The results of the interviews will inform the design of the prototype. Dependency management will be solved using the service locator pattern and dependency injection where possible, with the view to use composition over inheritance where possible.

C++ is a good language for the prototype according to participants, and personal experience with it is sufficient enough to develop the application.

The network update rate will be made easily configurable, and will be starting at 30 updates per second as a basis.

In terms of threading, networking will be performed in a separate thread, however for the time being all other operations will be run in the same thread. This will be changed if the systems begin to become a bottleneck for other systems, however – for example, if physics simulation starts to slow down game logic.

5 Design & Development

5.1 Version Control

Source control is crucial for keeping the history of files when they are changed, so changes can be reversed at any time. Subversion will be used for this project, since it consists of one developer, and version control software can be changed at any time with little downtime and development effort. (Collins-Sussman, Fitzpatrick & Pilato 2004)

5.2 Build System

As the prototype will be using cross-platform code standards and structures, it will not be platform specific. Because of this, it requires a cross-platform build system capable of creating various project types for development environments on different operating systems.

CMake is an appropriate solution for this, as it is freely available, and can be used for various development environments. (CMake - Cross Platform Make 2013) Similar to the version control system selected, this has little bearing on the structure of the project and can be changed at a later date without re-writing any code.

5.3 Development Environment

The development environment of choice for this project is Microsoft Visual Studio, available via the University, as discussed in section 2.1.2. A plethora of other tools are available, but Microsoft Visual Studio has been used before, so is a good choice for this project.

It should also be noted that any other IDE can be used without any code changes, if no Microsoft specific extensions are used. As the prototype is designed to be cross-platform, only standard C++ extensions will be used.

5.4 Self-Documenting Code

In order to ensure the prototype is accessible as possible to prospective clients, focus will be given to ensuring that code is easily understandable when it is looked at.

For instance, descriptive method and variable names are preferred, and code will be logically ordered into separate classes where appropriate. Comments will also be used for complex sections of code.

5.5 Designing an Engine

As the code is designed to be oriented around a client/server architecture, it needs to be split into three main areas:

- Client code – code that is never run on the server
- Shared code – code that is run on the server and the client

- Server code – code that is never run on the client

This means that the code will use two projects: Client and Server. Shared code will be the same files included by both projects.

Both projects will require an executable of their own in the beginning – the suggested dependency chain is shown in Figure 1.

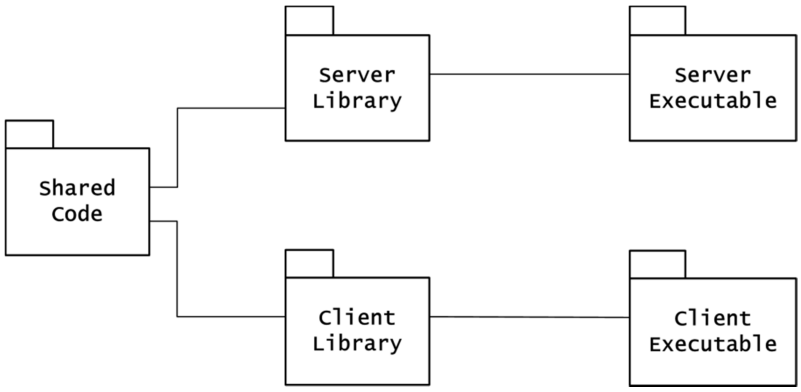


Figure 1 - Diagram of the project layout when running in dedicated mode

This project layout allows for the server to run in two modes – "dedicated" and "local". Dedicated mode would allow for the client executable to house the client project, and the server executable to house the server code only.

Local mode would instead replace the two executable projects with a single executable, allowing a single executable to run both the client and server on the same machine. This could be used for single player games, whereas dedicated mode could be used for multiplayer games.

A hybrid local and dedicated mode could also be used, allowing both single player and multiplayer games by including a client executable with both the server and client, and a dedicated server executable containing only server code.

As these changes are relatively simple to make, they can be left to the discretion of the game developer. During development, the projects will remain split up as in Figure 1 to offer a healthy level of separation.

5.5.1 Class Structure

As described in Figure 1, the bulk of code for the application will be contained within the shared, client and server projects. The below sections give outlines of their overall structure.

5.5.2 Shared Code

Appendix 13 shows the proposed class diagram for the shared code portion of the project. It shows shared systems, and the inheritance chain between game objects. Some inheritance lines have been removed, for instance classes implementing interfaces `IRendered` and `ISimulated` are not included to simplify display.

Composition has been used to construct the diagram, with various interfaces used instead of flat inheritance. This was suggested as part of the interview research, and was implemented based on the discussions in section 4.3.5.

The shared code includes various services that the code will consume – for instance, a `WorldManager` class is included, as both the client and server will need to have a central location to manage `BaseGameObject` classes. A service locator has been designed (also due to the discussion in 4.3.5) as `Locator`. Network services are also encapsulated in the shared code, using the `Communicator` class.

The engine recognises two main types of `BaseGameObject` – geometry and entities. Geometry, described by the implementation of `BaseRenderedGeometry` and `BaseSimulatedGeometry`, allows for non-moving set pieces that are described using the `Polygon` class, which contains a list of `Point` structures needed to construct a convex shape.

Similarly, entities are encapsulated in `BaseSimulatedEntity`, a class implementing `IRendered` and `ISimulated` (composition). A shared "player" class, `BasePlayerEntity`, inherits `BaseSimulatedEntity`.

The code also includes utility data types such as `Point`, `Color` (American spelling to keep consistent with libraries written by American developers), `Vector` and `Polygon`. These structures will overload basic operators for convenience, so code such as `Point + Point` will yield a correct and intended result.

5.5.3 Client Library

The client library will include the shared code base as a starting point, to take advantage of the pre-written classes such as `Point`, and so the same data structures can be shared between the server and the client (such as `BaseGameObject`).

The client library class diagram can be found in Appendix 14. The diagram shows a simple UI system, starting with the View class and some simple visual controls inheriting from it. Since this is a prototype, these features may not be implemented, but the `WorldRenderer` will, as it will allow the scene to be drawn. Therefore, `WorldRenderer`, `MainView`, and `View` must be implemented in the prototype.

Moving down the diagram in Appendix 14, various other classes can be observed. Some client-specific services exist, including `SFMLDrawing`, `WindowManager`, `InputManager`, `ClientUpdateManager`, and `ClientFactoryManifest`. These classes offer specific functionality pertaining to drawing objects on screen, managing the operating system window, accepting input, accepting client updates, and providing a factory service to create game objects from level files.

`ClientPlayerEntity` and `ClientPhysicsEntity` also exist, both implementing `IClientNetworked` and other specific shared classes pertaining to their functionality – in the case of the player entity, the shared `BasePlayerEntity`.

5.5.4 Server Library

In the same manner as the client library, the server library will include shared code directly. A class diagram showing the server code only can be found in Appendix 15.

Services included on the server are `WorldSimulator`, `ServerUpdateManager`, and `ServerFactoryManifest`. The simulator encapsulates an instance of the Box2D library, and provides physics simulation for the engine. The update manager is responsible for updating and dropping clients, and accepting connections. The factory manifest operates in exactly the same as the `ClientFactoryManifest` discussed in 5.5.3, implementing the factory pattern to create game objects from level files.

In a similar fashion to the client library, various objects inherited from the shared code are present – for instance, `ServerPlayerEntity`, `ServerPhysicsEntity`, `ServerStaticGeometry` and `ServerStaticEntity`. This allows the execution of server-specific code for these game objects, such as physics simulation code (simulation is performed server-side only).

5.5.5 Network Code

One of the goals of this project is to compare the use of TCP and UDP in a multiplayer game engine. This will need careful consideration in code, so that the engine is able to use both TCP and UDP transparently.

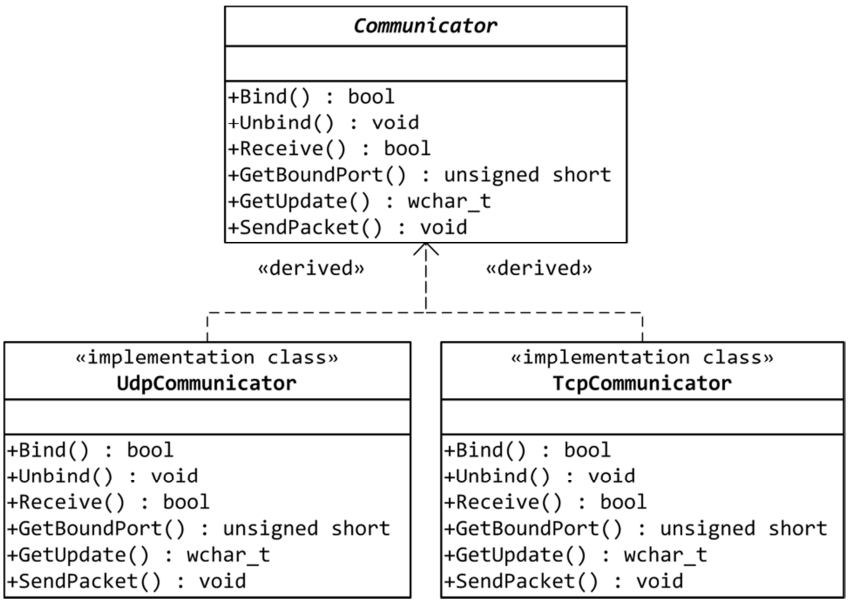


Figure 2 - Class Diagram Showing the TCP and UDP Communicator Classes

The proposed structure will allow the code to recognise one abstract class, `Communicator`, and for that to be implemented using the derived classes `UdpCommunicator` and `TcpCommunicator`, each providing specific implementations of the UDP/IP and TCP/IP protocols respectively.

An implementation of this type allows the engine to use TCP or UDP as required without adjusting any of the code in the engine (other than code responsible for

initialising a `Communicator` class). For comparing the two technologies, this is a crucial feature.

5.5.6 Catering for Packet Loss

Due to the nature of possible network protocols used, packet loss must be factored into the classes managing updates in the prototype. UDP/IP (discussed in section 2.3.2) does not ensure reliable delivery of network packets, therefore, the client must have enough data to continue rendering the world.

The solution that will be used in the prototype for this is that each update sent from the server will include all possible data about each entity, so that if a packet is lost, the client will be able to interpolate the entity position with the last received packet.

```
24  void WorldRenderer::DrawRenderable(BaseGameObject *pEntity, IRendered *pRenderable)
25  {
26      auto oRenderables = pRenderable->GetRenderables();
27      for (auto pPoly : oRenderables)
28      {
29          if (pPoly->textureReference < 0)
30          {
31              Debug::DebugMessage("Loading texture %", pPoly->texturePath);
32              pPoly->textureReference = Locator::Drawing()->LoadTextureResource(pPoly->texturePath);
33          }
34
35          const float dumbSmoothFactor = 0.75f;
36
37          pRenderable->lastPosition = (pRenderable->lastPosition * dumbSmoothFactor)
38              + (pEntity->position * (1.0f - dumbSmoothFactor));
39
40          pRenderable->lastRotation = (pRenderable->lastRotation * dumbSmoothFactor)
41              + (pEntity->rotation * (1.0f - dumbSmoothFactor));
42
43          Locator::Drawing()->SetTexture(pPoly->textureReference, pPoly->fill);
44          Locator::Drawing()->DrawPolygon(pPoly->polygon, pRenderable->lastPosition,
45              pRenderable->lastRotation);
46      }
47  }
```

Figure 3 - "Dumb" Client Rendering Interpolation to account for packet loss

Figure 3 shows the final version of the code. The smoothing is referred to as “dumb” smoothing, as it only serves to smooth the position and rotation of rendered entities, it does not offer latency compensation.

$$position = (last\ position * 0.75) + (received\ position * 0.25)$$

Equation 1 - "Dumb" smoothing equation

Equation 1 shows a simplified version of the algorithm for illustration purposes. It can be seen that 25% of the received position vector is taken, and is added to 75% of the old position. The result is a very cheap way of smoothing game objects as they are rendered on-screen.

5.5.7 Main Loop

All game engines run a loop, allowing systems to update each time the loop is passed. A main loop structure that could be used for the client and the server is shown in Table 3.

Client.cpp	Server.cpp

while (running)
{
ReceiveNetworkUpdates();
 	
ProcessInputEvents();
 	
RenderFrame();
 	
SendUpdates();
}
	while (running)
	{
	ReceiveNetworkUpdates();
	
	SimulateWorld();
	
	SendNetworkUpdates();
	}

Table 3 – Proposed main loop for the client and server projects

The client is responsible for processing updates, accepting user input, rendering a frame and sending the user input to the server, and the server is responsible for processing updates, simulating the world from those updates and sending updates back to all clients.

5.5.8 Configuration Files

The engine will need a configuration file format to allow data structures to be described using text files. This enables the engine to adjust the way it operates without requiring it to be re-compiled, and also allows objects and levels to be recorded in a file, to be edited and iterated.

There are a few file formats capable of this task. Two formats will be explored below, with a common piece of example data to demonstrate their expressiveness.

5.5.8.1 XML – Extensible Mark-up Language

A format that uses tokens encapsulated in angled brackets (< >) to describe data structures. The structures can include any data types, as long as they are valid XML. Any characters that are part of the XML specification (quotes, angled brackets etc.) have to be carefully handled so that parsers do not assume they are part of the XML structure itself.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <level>
3    <name>Test Level</name>
4    <objects>
5      <tree position="10, 0"></tree>
6      <plant position="40, 0"></plant>
7    </objects>
8  </level>
```

5.5.8.2 JSON – JavaScript Object Notation

JSON (often pronounced Jason) uses data structures available in JavaScript to store data. The specification is more rigid than XML, that is, only a select few data types are available, and all JSON parsers are required to understand them. (ECMA International

2013) JSON is widely used in web development, as it is a more compact format than XML, and integrates well into JavaScript engines available in web browsers. JSON is less-human readable than XML, however.

```
{
    "level": {
        "name": "Test Level",
        "objects": {
            "tree": { "position": [10, 0] },
            "plant": { "position": [40, 0] }
        }
    }
}
```

5.5.8.3 INI – Initialisation Format

INI format, or "initialisation format" is a common format used for storing configuration data for applications.

```
[level]
name=Test Level
objects.tree.position = 10, 0
objects.plant.position = 40, 0
```

The INI format does not support "arrays" of data like XML and JSON do, so such a concept has to be created, and the INI parser must be extended to accept the array format. This means that INI files will become very bulky when dealing with multiple hierarchies of data, and editing the files becomes confusing and cumbersome.

5.5.8.4 Chosen Format

The format that will be used to structure configuration files for the engine will be JSON. This is because it offers good portability – for example, all that is needed to read the data is a standard JSON parser, however with XML the code needs to have prior knowledge of the format before it can read the data. INI format is not expressive enough to be used, as it does not natively support array structures, and is an informal data format.

JSON configuration files will be used to dictate engine configuration files, level files and any other files that describe data structures in the engine.

5.5.8.5 Engine Configuration Parser

To parse JSON files, the project will use JsonCpp, an open-source library written in C++, offering object-oriented accessing of JSON data. (Lepilleur 2014)

To ensure that this specific system can be changed, and isn't dependent on its implementation library (JsonCpp), a wrapper class will be created called `Manifest`. The class diagram for Manifest is included below.

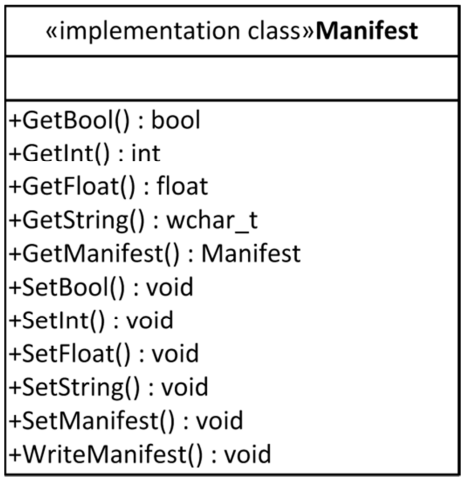


Figure 4 - Manifest Class Diagram

This encapsulation class allows for setting and retrieving data types needed by the engine, without exposing the engine to the underlying data structure (JSON).

5.6 Add Unit Tests

Unit testing allows for testing "units" of code, such as a function, or a class. Unit tests are typically run every time the code is built, or before changes are added to the central code repository (discussed in section 5.1).

With the chosen language, C++, the unit testing is performed with unit testing libraries. There are two popular libraries available:

- CppTest - http://cpptest.sourceforge.net/
- googletest - https://code.google.com/p/googletest/
- Visual Studio Built-in

Both with very similar design. CppTest does not appear to have proper support with the chosen developer environment (see section 5.3), however a free extension for googletest exists, offering full integration. (Lindqvist 2014)

Visual Studio's built-in test suite offers the best integration with code, with full support from Microsoft. (Microsoft Corporation 2013d) Because of the support inside the developer environment, the built-in library will be used for unit testing the prototype as necessary.

5.6.1 Areas Requiring Unit Testing

If the prototype were an actual engine, the entirety of the project would arguably require unit testing. However, since this is a prototype application, unit testing will be performed on specific areas only:

- Network code – testing for the communicator classes
- World manager – testing for the class that manages all game objects

5.7 Barebones Code Structure

To begin the code, C++ header files will be utilised to author the "barebones" code structure first, according to class diagrams.

C++ header files allow for the separation of the declaration and implementation of variables and methods. (Stroustrup 1997, p. 27)

Appendix 16 shows the declaration of the `Manifest` class, which can be later implemented using the declaration.

Such an implementation allows for other header declarations to reference the `Manifest` class with support of the development environment's syntax highlighting and code suggestions.

5.8 Fleshing Out The Code Structure

This stage of development involves writing methods based on the header declarations, and connecting the application together.

5.9 Adding a Basic Renderer

Rendering the objects in the world is one of the most important aspects of an engine, and the complexity of rendering systems can exceed that of any other class in the engine. The prototype will include a simple renderer, with support for rendering 2D objects and 2D text on-screen.

5.9.1 Structure

The renderer's structure will fit with that of the `View` classes, discussed in section 5.5.3. This means that it will at least have a `Draw()` method that will be called every frame, and the renderer will update the screen accordingly.

5.9.2 Implementation

The draw function will first need to obtain an up-to-date list of all objects available in the world, and then iterate through that list, drawing each entity at its current position.

The code will be split into sections – the code responsible for looping entities, and the code responsible for the drawing of each entity, once it has been established as an entity that needs to be rendered. Rendered entities implement the `IRenderable` interface, so can be distinguished from non-rendered objects using the C++ `dynamic_cast` operator.

Dynamic casting allows a variable to be interpreted as different type at runtime, which allows child classes to be changed to their parent classes or interfaces. (Stroustrup 2011, p. 130)

Appendix 17 includes the full implementation of the Draw() function. It shows that a list of rendered objects is obtained from the `WorldManager` using the Service Locator.

5.10 Testing the Prototype and Fixing Bugs

This is a continual point of interest during the development of the prototype, as it represents an on-going effort during the development phase of the project. Bugs (sometimes comically referred to as “unintentional features”) are the name given for parts of the application which behave unexpectedly, leading to undesired behaviour and in some cases crashing.

5.10.1 Continual Testing

During development, focus has been given to attempting to thwart unintended issues with continual testing. Each code change meant that the application was started and tested to make sure everything remained in a working state, with the view of catching any regression issues before they arose.

5.10.2 Assertions

Figure 5 shows an assertion in the `WorldManager` class. An assertion is a check performed at run-time that a certain statement is true. It is akin to saying “I assert that this value is true”, if it were to be taken as a spoken sentence.

```
74  BaseGameObject* WorldManager::CreateEntity(Manifest oManifest)
75  {
76      auto szEntityManifestType = oManifest.GetString("type");
77      auto pGameObjectFactory = Locator::FactoryManifest()->GetFactory(szEntityManifestType);
78      assert(pGameObjectFactory);
79      auto pEntity = CreateEntityFromFactory(pGameObjectFactory, oManifest);
80      return pEntity;
81  }
```

Figure 5 - An assertion in the WorldManager

While running with the application in debug mode, if the assertion doesn’t pass, a message is shown (Figure 6).

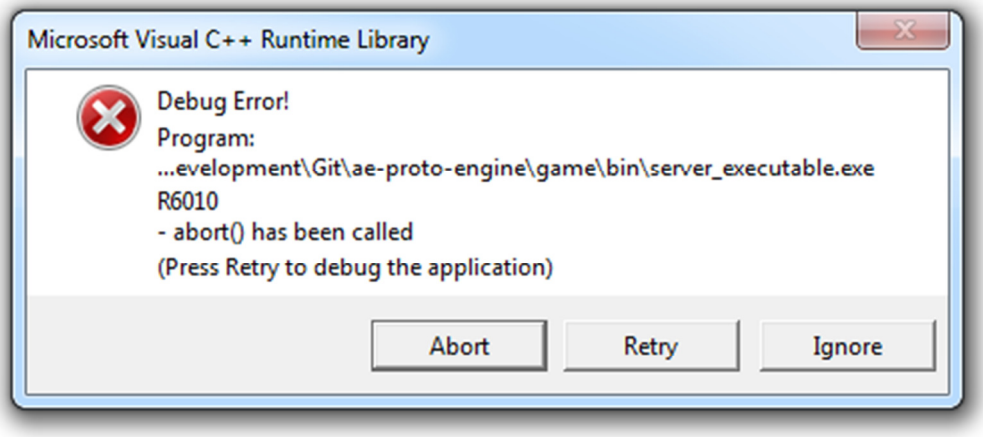


Figure 6 - The result of a failed assertion in debug mode

The message allows the programmer to further investigate the issue.

6 Implementation & Testing

To make sure that the prototype meets its objectives and functions appropriately, testing must take place. Different forms of testing are described in this section, and their implications for the prototype.

6.1 Unit Testing

The implementation details of unit testing are described in section 5.6. Unit testing allows for portions of code (or "units") to be tested in a sandboxed setting. (Microsoft Corporation 2013e) An example unit test is included as Appendix 18.

The code shows an example class with a method that returns a positive result. The unit test makes a new copy of the class, and asserts that the result of the method is positive. If it is, the test passes, and if it isn't, the test fails. An example of the results window that can be seen in Visual Studio when such a test is run is included below the code excerpt.

Since the scope of this project is a prototype, there is not enough time to implement unit tests for code in the project. Therefore, focus will be put on other forms of testing.

6.2 Performance Analysis

Performance analysis is a feature of Microsoft Visual Studio, allowing developers to observe portions of code which serve as speed bottlenecks. (Microsoft Corporation 2013f)

This feature is provided with Visual Studio, the IDE of choice for development of the prototype.

6.2.1 Hypothesis

Expensive areas of the application should include code which is running expensive calculations, and performing memory allocation. This is logical as the processor has to spend more time executing these parts of the program, as they're more complex.

Other areas that are "expensive" are portions of code that have to wait for a device, such as network operations. Therefore, it can be assumed that the following subsystems will be shown as the most processor intensive portions of the engine while it is running: rendering, physics simulation, network operations and file operations.

6.2.2 Data

Appendix 19 shows the results of a performance analysis session on the server project, with two clients connected to the server. The report seems normal, and confirms the initial hypothesis – `Simulate()`, the physics simulation call and `SendUpdates()` / `ReceiveUpdates()` are the bottlenecks in this case, which is expected.

Appendix 20 shows the results of a similar session for the client project. The results differ slightly from the given hypothesis, as network operations seem to be very inexpensive in comparison to drawing calls. In fact, the rendering code seems to be expensive when converting polygons from the engine's `Polygon` class to the SFML `ConvexShape` class (SFML is discussed in section 2.2.2).

Appendix 21 shows a further drill-down into the code in question causing bottlenecks. It can be observed that the code looping through points in the polygon, and adding them to an SFML polygon is the most expensive call, with 22.6% of the CPU time spent inside the function used on the `oShape.setPoint()` call.

This hints at an area for optimisation, and as a piece of further work, suggested methods of making this code faster are needed in order to make the application as fast as possible.

6.3 Network Packet Tracing

To check the contents of network packets on the fly, a network packet inspector may be used. A freely-available tool called *Wireshark* is able to capture and inspect packets, and includes a graphical interface for filtering and sorting traffic. (Wireshark Foundation 2014)

Appendix 22 displays a packet trace result from the server project, and selected is one of the packets sent from the server to the client when a connection is made. The name of the level, `simple_level.json` can be seen in the bottom portion of the screenshot.

This packet is the first formal communication sent from the server to the client, and it can be observed by the entries below it that the data is not just sent once, it is sent multiple times (evidenced by the packets also 71 bytes in size, sent in the same time frame).

This is an area that needs to be reviewed in further work, as it could potentially be a waste of bandwidth to send the same data several times.

6.4 Dealing with Firewalls

Most of the devices connected to the Internet do so using the Internet Protocol version 4 (IPv4). (Tanenbaum 2011, p. 439) Therefore, providers and users have made provisions to use the limited address space as efficiently as possible. The most prevalent solution to the problem is Network Address Translation (NAT). (Tanenbaum 2011, p. 451)

NAT (Network Address Translation) allows multiple hosts to connect to a network (such as the Internet) using one public-facing IPv4 (Internet Protocol version 4) address.

This poses a problem for the engine, as once a client connects to the game server, the server must begin sending packets to the client. With NAT, this is made difficult, as clients are not exposed.

The solution to this problem is referred to as “hole punching”. Hole punching allows for server applications to send data back to the port the client sent data from, and as a mapping has been created by the client’s gateway device for this port, the connection is successful. (P. & Egevang 2001)

The implications of NAT traversal for the prototype engine are that the server must send data to the client on the port that it received data on. The code snippet in Figure 7 shows how the server deals with NAT traversal when sending updates to clients.

```
56 void ClientUpdateManager::SendUpdates()
57 {
58     int updateType = 0;
59     if (!(m_oLastReceivedUpdate.data >> updateType))
60     {
61         CommunicatorUpdate_t update;
62         update.data << CLIENT_UPDATE_INITIAL;
63         update.host = m_oRemoteHost;
64         m_oLastSentUpdate = update;
65         m_pCommunicator->SendPacket(update);
66     }
67     else
68     {
69         CommunicatorUpdate_t update;
70         update.data << CLIENT_UPDATE_FULL;
71         update.data << Locator::InputManager()->SampleInput();
72         update.data << Locator::InputManager()->SampleMousePosition();
73         update.host = m_oRemoteHost;
74         m_oLastSentUpdate = update;
75         m_pCommunicator->SendPacket(update);
76     }
77
78     // Bind to a port to
79     // listen to data returning
80     BindToOutgoingPort();
81 }
82
83 void ClientUpdateManager::BindToOutgoingPort()
84 {
85     // If the local port has changed
86     // (or isn't 0 anymore, initial update)
87     // then listen to it (NAT traversal)
88     CommunicatorPort_t oBoundPort = m_pCommunicator->GetBoundPort();
89     if (oBoundPort != m_oLocalPort)
90     {
91         m_pCommunicator->Unbind();
92         m_pCommunicator->Bind(oBoundPort);
93         m_oLocalPort = oBoundPort;
94     }
95 }
```

Figure 7 - The prototype's NAT traversal solution

Lines 89 to 94 can be observed to be binding to the port that the client last sent data from, if that port has changed. This enables the server to communicate with the client when a firewall and NAT functionality is present.

6.5 Configuration

One outcome from the interview process was the need for a varying update rate for the prototype. In section 4.3.7, it was concluded from candidate responses that different games require different update rates.

The prototype includes a facility to change this using its configuration feature, discussed in section 5.5.8.

```
{
    "client_antialias" : 6,
    "client_icon" : "client_icon.png",
    "client_size" : [ 802.0, 587.0 ],
    "connect_host" : "127.0.0.1",
    "connect_port" : 1236,
    "connect_protocol": "udp",
    "debug" : false,
    "debug_font" : "inconsolata.otf",
    "fps_limit" : 60,
    "level_directory" : "../levels",
    "model_directory" : "../models",
    "update_rate": 30, // Number of times to send out updates a second
}
```

Figure 8 - Example Client Configuration – client_manifest.json

Figure 8 shows the server manifest, the file responsible for settings on the server. The `update_rate` and `listen_protocol` settings are of particular interest; they allow for changing the number of update packets sent out by the client each second, and switching the protocol used by the client (TCP or UDP).

The file also includes specific client configuration such as the level of anti-aliasing for the renderer, the server host and port to connect to, and whether to run in debug mode.

```
{
    "level_directory": "../levels",
    "model_directory": "../models",

    "listen_port": 1236, // Listen settings
    "listen_protocol": "udp",
    "level_name": "simple_level.json", // Level to start on
    "client_timeout": 10, // Client timeout is 10 seconds
    "update_rate": 60, // Number of times to send out updates a second
    "simulation_rate": 30 // Number of times to simulate per second
}
```

Figure 9 - Example Server Configuration – server_manifest.json

Similarly, Figure 9 shows complimenting settings for the server, with the added field for `simulation_rate`. Other fields exist, namely to set the level, the port the sever is listening on, and on-disk paths of the level and model files.

6.6 Viability of TCP/IP and UDP/IP

This section aims to discuss the viability of each technology in terms of a multiplayer networked game engine.

6.6.1 Nature of the Game

The series of Interviews conducted highlighted some important points, namely that the answer to the most viable technology isn't black and white. The interview conclusions in section 4.3.7 highlight the need for the frequency of packets to be dependent on the nature of the game itself.

However, this does have an impact on the protocol used by the engine. As discussed in section 2.3, TCP offers reliable delivery of packets at the cost of speed, and UDP offers unreliable delivery of packets at a much faster rate.

Therefore, the protocol used is also dependent on the number of updates needed per second – if the frequency of updates is low, TCP is a perfectly viable protocol for use in the engine. If a high number of updates is desired, UDP becomes the better choice, but at the cost of reliability.

To that end, both protocols could be viable for a game engine, it simply depends on the nature of the game.

6.6.2 Considerations for TCP/IP

As discussed in section 2.3.1, TCP is not as fast as UDP, however offers reliable delivery. In terms of the prototype, consideration must be given for packet flow, not just update speed, in order to prevent low-bandwidth clients from being bombarded with packets they are unable to process. (Comer 2005, p. 136)

6.6.3 Considerations for UDP/IP

If UDP is used for games built using the prototype engine, the one major pitfall they must accommodate is packet loss. This means that systems in the engine must be designed in such a way that transmitted data can simply be lost, and subsequent updates sufficient for providing a smooth experience, or the engine keeps track of which packets have been received. This would involve simulating the core functionality of TCP: redundancy. However, instead dealing with it at the transport layer, it would be dealt with on the application layer. (Zimmermann 1980)

6.6.4 A Hybrid Approach

Future work may involve the investigation of a hybrid approach to networking to solve issues prevalent in avenues explored above, using the best assets of both protocols for data transmission and receipt.

One such approach may involve transmitting non-sensitive data (such as positions, rotations, bearings, velocity data) via UDP/IP, and using TCP/IP for crucial updates,

such as game object creation, new/disconnecting clients, and in-game chat applications.

Such an approach would mean that the prototype game code would need explicit awareness of the protocols in used by its network communication classes, breaking encapsulation. This approach would require the prototype's network core to be re-written, as it was engineered to allow separation of game code and network code (discussed in section 5.5.5).

7 Evaluation

To examine whether the project was a success, the initial requirements must be closely examined against the entire project. The original project objectives are included in the project specification, which can be found in Appendix 1. Each objective and how well it was met is discussed below.

The quality and usefulness of the application will also be discussed, using appropriate metrics to measure the achievement of each objective.

7.1 Existing Software, Methods and Tools

Three related objectives around this area have been grouped into the headings below, indicating that they were explored in the same section.

7.1.1 Research and Compare Existing Software and Development Methods

This objective was explored in section 1. It included sections about Programming Languages (2.1), Graphics, Input and Sound Libraries (2.2), Communication Protocols (2.3), Physics Libraries (2.4), Similar Works (2.5) and Development Methodologies (2.6).

These sections offer a thorough account of each area of development research. Each section first offers an overview of each technology found to be relevant to research in the area. This typically includes a history of the item, and a list of features, drawbacks and limitations present withing the technology.

The metric that can be used to see whether this objective was explored successfully can be seen in section 1.3.1, in the introduction. The section states:

> *The metrics for measuring the completeness of this objective can be seen in the research performed in this document.*

It can be observed that the research in the document shows that this objective was completed successfully – section 2 clearly shows that each existing software and development method was thoroughly researched.

7.1.2 Compare Each Method and Tool, Evaluating Drawbacks and Merits

Further sections offer a comparison between each technology, based on the research about which to use, based on the requirements of the prototype and any future work.

The metrics for measuring the completeness of this objective are listed in section 1.3.2.

> *Metrics for this objective can be measured as the inclusion of an extensive analysis and comparison of each method and tool described.*

The metric can be used to prove that the objective has been thoroughly explored by looking at section 2. The closing parts of each section offer extensive comparison of

each technology. For example, this is evidenced in section 2.1.5 when comparing programming languages, and subsequently in other sections.

However, some areas of this comparison could be better – for instance sections on development methodologies (2.6) and communication protocols (2.3) do not contain as thorough comparisons as some other tools (see 2.2.5).

7.1.3 Decide Which Tool to Use, and How to Use it

The final section of each piece of research includes a final analysis, offering an answer about which tool should be used in the prototype. This goal has been successful, as the tools have then been used to develop the prototype in the development section.

The metric used to measure the completeness of this section is listed in section 1.3.3:

> *The metric for measuring the success of this objective in the final project can be observed as a final decision about each tool and technology, based on existing comparisons and research.*

Using this metric, it can be observed that the selected technologies described in section 2 such as SFML, Box2d, C++ are used in the final prototype. The research performed in section 2.5 can also be seen to have had an effect on the final design. In the section, the Source Engine's client and server architecture is analysed; this can be seen to have had an impact on the final design of the project, as in section 5.5, the prototype also uses a client and server project architecture.

7.2 Conduct Interviews With Knowledgeable People

This objective was completed in section 4, and its results used in future sections during the actual development of the prototype.

Interviews were conducted with five people – two people currently working in the games industry, two people currently working in software development with experience in the industry, and one lecturer with experience as a project lead at games company.

The interviews were performed successfully, and from the results it was felt that follow up questions weren't needed, as each candidate responded as required for the development of the project.

Results were collated and analysed in section 4.3, offering conclusions about how each result would impact the project. As a measurable result, the project had no specific method to handle inter-application dependencies before the interview process. However, candidates identified several patterns to explore, including the Service Locator pattern, the method used in the prototype's development phase.

The metric showing whether this objective was achieved is described in section 1.3.4:

> *Metrics for this objective include the conduction of interviews with selected knowledgeable candidates.*

Using this metric, it can be observed that this objective has been completed, as the interviews were conducted with knowledgeable people (see section 4.2).

7.3 Create an Engine Design Based on Existing Technologies

This objective describes the process of designing an engine based on existing technologies. In this case, the technologies were explored and selected in the primary and secondary research phases, and collated into a design plan in this section.

Evidence for this objective can be observed in the design itself; each phase includes researched elements. An area influenced the most by existing technologies is the architecture of the engine itself – the project is split into a client and server project, based on research performed into existing technologies in 2.5.1.

Metrics for measuring the achievement of this objective can be seen in section 1.3.5:

> *The metric for measuring the completeness of this objective can be seen in the final prototype design, which will be based on existing research and technologies.*

As mentioned above, the design chapter has been impacted by the research, as discussed in sections 2.7, and 4.5.

7.4 Implement, Test and Document a Multiplayer Game Engine Concept

This objective is explored in section 6. The section documents the various methods of testing performed on the prototype, and how the results impacted the project. The "Multiplayer Game Engine" implication of this engine is evidenced by the final product, which can support multiple players from remote networks across the Internet.

Implementation details of the prototype are covered in the previous section (section 5.7 onwards), which includes high level overviews of the implementation of some systems in the prototype, and code snippets of more complex systems that require more explanation and exploration.

However, documentation is lacking for the prototype. This has been due to a lack of time, with more time focussed on development rather than documenting the prototype engine design for developers in the future.

The metric used for measuring the completeness of this task is defined in section 1.3.6:

> *The metric that will be used for measuring the achievement of this objective will be to see if a multiplayer game engine prototype is created, including appropriate implementation, testing and documentation.*

The implementation can be seen to have been achieved by looking at the final version of the prototype code. The testing of the prototype can also be seen in section 6, which shows that the application does as it was designed to do.

As mentioned above, documentation can be seen to be lacking in some areas – in fact the only formal version of documentation can be seen in the comments on different areas of code in the prototype. This objective has been achieved, but it is evident that more work needs to be undertaken in this area.

7.5 Consider Different User Limitations Throughout, Such as Packet Loss and Latency

This objective has not been met to its fullest extent, however has been explored to some degree. Packet loss is not an issue for the engine with its current design, as the server portion of the prototype caters for packet loss using methods described in section 5.5.6.

The implementation discussed describes how packet loss does not impact the engine and offers a simple smoothing technique, however doesn't discuss latency compensation. This is an area requiring further work if the prototype is to be suitable for real-world scenarios.

The metric to analyse the completeness of this objective is defined in section 1.3.7:

> *The metric for analysing the completeness of this objective can be seen throughout the report, and finally in the implementation of the prototype, which should cater for packet loss and latency.*

The packet loss portion of the objective has been explored specifically in section 5.5.6 (as mentioned above), however only a portion of the objective has been explored.

Further work would involve exploring the effects of latency on the prototype, and how to reduce its effects.

8 Conclusion

8.1 Summary

In order to reflect on the project and offer a conclusion about its content, it is first wise to take a step back, and look at the project's initial goal. The below is an excerpt from the project specification, included as Appendix 1.

> *The project will explore multiplayer networking techniques in a modern game engine. Its focus will be comparing two different networking technologies – TCP/IP and UDP/IP, and discuss the viability of each in terms of networking games. It will explore topics such as packet loss, limitations on clients such as bandwidth constraints and the intervention of technology such as firewalls.*

The viability of each communication technology has been discussed throughout the project, with initial research (section 2.3), implementation into the design (section 5.5.5) and discussions surrounding the viability of each technology (section 6.5).

The results for each stage have been clear; both protocols perform well at some tasks, and badly at others. The overall conclusion has been that TCP simplifies game code (as it takes are of packet acknowledgement) at the cost of speed, and UDP allows for the speed of delivery over reliability of receipt.

There are several issues surrounding the protocols that need to be taken into account, such as frequency of updates, and the nature of the game itself.

8.2 Recommendations

One important aspect of reflection is to ask what others should know if they are to embark on a similar project.

1. *Construct a prototype using existing libraries, it is not a wise use of time to write existing code from scratch*
2. *When writing code that consumes libraries, creating a layer on top of the library is advised so it can be swapped out at a later date*
3. *Dependency injection and the Service Locator pattern are good methods of handling dependencies within code*

Recommendation 1 describes a common-sense approach to constructing a prototype. There have been several instances in the development of the prototype where complex services were required. The physics library is the best example of this (described in section 2.4) – to construct a physics library from scratch involves a lot of complex mathematics, with great room for error. Using a library meant time could be spent in other areas of the prototype.

Recommendation 2 refers to the efforts used in the prototype to hide the implementations of third party components behind more generic interfaces for consumption by the application. This is important, as it means third party components

can be swapped out at any time – for example in response to license restrictions, changes in functionality, or issues encountered with the third party code. A good example of successes with this method can be seen in section 5.5.5, which shows socket code wrapped up as engine-specific classes `UdpCommunicator` and `TcpCommunicator`.

Recommendation 3 describes a point that was a focus of the research performed before the development of the project – how to handle inter-class dependencies. The conclusion drawn from interviewees was to explore dependency injection and the Service Locator pattern (conclusions from section 4.5). These techniques have been used to much success in the prototype, allowing different components to easily talk to one another, without using tightly coupled code. This also facilitates recommendation 2, as tightly coupled code cannot be easily changed.

8.3 Further Work

For further work in this area, an extension of the work in this project would be to create an engine that has some awareness of the features of its network protocols, using the assets of both to transmit and receive data (discussed in section 6.6.4).

To extend the engine, it should also be considered for 3D usages, a trend set by the current gaming marketplace. To equip the engine for 3D, the renderer must undergo the biggest changes. The engine code itself is very non-specific with regards to data it carries, so converting game code would involve changing `Point` instances (a vector of two values) to `Vector` instances (a vector of 3 values).

9 Table of Figures

Figure 1 - Diagram of the project layout when running in dedicated mode 30
Figure 2 - Class Diagram Showing the TCP and UDP Communicator Classes 32
Figure 3 - "Dumb" Client Rendering Interpolation to account for packet loss 33
Figure 4 - Manifest Class Diagram 36
Figure 5 - An assertion in the WorldManager 38
Figure 6 - The result of a failed assertion in debug mode 38
Figure 7 - The prototype's NAT traversal solution 41
Figure 8 - Example Client Configuration – client_manifest.json 42
Figure 9 - Example Server Configuration – server_manifest.json 42

10 Table of Tables

Table 1 - Functional analysis for programming languages 4
Table 2 - Functional analysis score table for libraries 9
Table 3 – Proposed main loop for the client and server projects 34

11 References

Activision 2013, *Call of Duty: Ghosts Leads Next Gen Launch*, viewed 12 March 2014, <http://investor.activision.com/releasedetail.cfm?ReleaseID=809656>.

Ambler, SW 2014, *Examining the Agile Manifesto*, viewed 12 March 2014, <http://www.ambysoft.com/essays/agileManifesto.html>.

Beck, K, Grenning, J, Martin, RC, Beedle, M, Highsmith, J, Mellor, S, Bennekum, A, Hunt, A, Schwaber, K, Cockburn, A, Jeffries, R, Sutherland, J, Cunningham, W, Kern, J, Thomas, D, Fowler, M & Marick, B 2001, *Manifesto for Agile Software Development*, viewed 12 March 2014, <http://agilemanifesto.org/>.

Bryman, A 2001, *Social Research Methods*, 2nd edn, Oxford University Press, New York.

Catto, E 2013, *About | Box2D*, viewed 13 January 2013, <http://box2d.org/about/>.

CMake - Cross Platform Make 2013, viewed 04 March 2013, <http://www.cmake.org/>.

Collins-Sussman, B, Fitzpatrick, BW & Pilato, CM 2004, *Version Control with Subversion*, O'Reilly.

Comer, DE 2005, *Internetworking with TCP/IP*, 4th edn, Prentice Hall, West Lafayette.

Digia Plc 2012, *Digia to Acquire Qt from Nokia - Digia Plc*, <http://qt.digia.com/About-us/News/Digia-to-Acquire-Qt-from-Nokia/>.

ECMA International 2006, *C# Language Specification*, <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf>.

ECMA International 2013, *The JSON Data Interchange Format*, viewed 06 March 2014, <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.

Electronic Arts 2011, *Battlefield 3 Launches With Record-Breaking Force - EA News*, viewed 12 March 2014, <http://www.ea.com/news/battlefield-3-launches-with-record-breaking-force>.

GarageGames, LLC 2013, *GarageGames - Torque2D*, <https://www.garagegames.com/products/torque-2d>.

Gomila, L 2013a, *License (SFML)*, <http://www.sfml-dev.org/license.php>.

Gomila, L 2013b, *SFML*, <http://www.sfml-dev.org/>.

Gottesdiener, E 1995, 'RAD Realities: Beyond the Hype to How RAD Really Works', *Application Development Trends*, pp. 28-38.

Gregory, J 2009, *Game Engine Architecture*, A K Peters/CRC Press.

Harbour, JS 2007, *Game Programming All in One*, 3rd edn, Thomson Learning Inc., <http://alleg.sourceforge.net/readme.html>.

References

Havok 2013a, *Havok Physics 2013*, <http://havok.com/sites/default/files/pdf/Havok_Physics_2013.pdf>.

Havok 2013b, *Customer Projects: Games | Havok*, <http://www.havok.com/customer-projects/games?product=Physics>.

Intel Corporation 2007, *Intel To Acquire Havok*, <http://www.intel.com/pressroom/archive/releases/2007/20070914corp.htm>.

Kohler, E, Handley, M & Floyd, S 2006, 'Designing DCCP: congestion control without reliability', *SIGCOMM '06*, vol 36, no. 4, pp. 27-38.

Kumparak, G 2011, *Creator Of Angry Birds' Physics Engine Calls Out Rovio For Not Giving Him Credit | TechCrunch*, viewed 13 January 2013, <http://techcrunch.com/2011/02/28/creator-of-angry-birds-physics-engine-calls-out-rovio-for-not-giving-him-credit/>.

Larman, C & Basili, VR 2003, *Iterative and Incremental Development: A Brief History*, <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1204375>.

Lepilleur, B 2014, *JsonCpp - JSON data format manipulation library*, viewed 06 March 2014, <http://jsoncpp.sourceforge.net/>.

Lindholm, T, Yellin, F, Bracha, G & Buckley, A 2011, *The Java® Virtual Machine Specification*, 3rd edn, Oracle America, Inc., California.

Lindqvist, M 2014, *GoogleTest Runner extension*, viewed 09 March 2014, <http://visualstudiogallery.msdn.microsoft.com/9dd47c21-97a6-4369-b326-c562678066f0>.

Microsoft Corporation 2013a, *Command-line Building With csc.exe*, <http://msdn.microsoft.com/en-us/library/vstudio/78f4aasd.aspx>.

Microsoft Corporation 2013b, *Fundamentals of Garbage Collection*, <http://msdn.microsoft.com/en-us/library/ee787088.aspx#what_happens_during_a_garbage_collection>.

Microsoft Corporation 2013c, *Multiple Platform Support | Microsoft.NET Framework*, <http://www.microsoft.com/net/multiple-platform-support>.

Microsoft Corporation 2013d, *Unit testing existing C++ applications with Test Explorer*, viewed 13 March 2014, <http://msdn.microsoft.com/en-us/library/hh419385.aspx>.

Microsoft Corporation 2013e, *Unit Test Basics*, viewed 14 March 2014, <http://msdn.microsoft.com/en-us/library/hh694602.aspx>.

Microsoft Corporation 2013f, *Beginners Guide to Performance Profiling*, viewed 14 March 2014, <http://msdn.microsoft.com/en-us/library/ms182372.aspx>.

References

Mihailescu, D 2010, *Benchmark start-up and system performance for.Net, Mono, Java, C++ and their respective UI*, <http://www.codeproject.com/Articles/92812/Benchmark-start-up-and-system-performance-for-Net>.

Oppenheim, AN 2001, *Questionnaire Design, Interviewing and Attitude Measurement*.

P., S & Egevang, K 2001, *RFC 3022 "Traditional IP Network Address Translator (Traditional NAT)"*, viewed 16 March 2014, <https://www.ietf.org/rfc/rfc3022.txt>.

Python Software Foundation 2013, *About Python*, <http://www.python.org/about/>.

Ritchie, DM 1993, *The Development of the C Language**, <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>.

Rossum, GV 2013, *The Python Language Reference*, 275th edn, Python Software Foundation.

Segal, M & Akeley, K 2013, *OpenGL 4.4 (Core Profile)*, Khronos Group, Inc.

Stroustrup, B 1997, *The C++ Programming Language*, 3rd edn, Addison-Wesley.

Stroustrup, B 2011, *An incomplete list of C++ compilers*, <http://www.stroustrup.com/compilers.html>.

Stroustrup, B 2013, *Bjarne Stroustrup's FAQ*, <http://www.stroustrup.com/bs_faq.html>.

Stroustrup, B 2013, *C++ Applications*, <http://www.stroustrup.com/applications.html>.

Sutherland, J & Schwaber, K 2013, *The Scrum Guide™*, <https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf>.

Tanenbaum, AS 2011, *Computer Networks*, 5th edn, Pearson Education, Inc.

Valve Software 2013, *Porting Source to Linux: Valve's Lessons Learned - YouTube*, <http://www.youtube.com/watch?v=btNVfUygvio&t=3192>.

Waveren, JMPV 2006, *The DOOM III Network Architecture*, Id Software, Inc.

Willison, F 2007, *An Interview with Guido van Rossum - O'Reilly Media*, <http://oreilly.com/pub/a/oreilly/frank/rossum_1099.html>.

Wireshark Foundation 2014, *Wireshark · About*, viewed 14 March 2014, <http://www.wireshark.org/about.html>.

Xamarin 2013, *Mono*, <http://www.mono-project.com/Main_Page>.

Zimmermann, H 1980, 'OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection', *IEEE Transactions on Communications*, vol 28, no. Apr 1980, pp. 425-432.

Appendix 1 Project Specification

PROJECT SPECIFICATION

PROJECT DEFINITION

Student: Alan Edwardes

Date: 21st October 2013

Supervisor: Dr. Keith Burley

Level of Project: BSc (Hons) Computer Networks

Title of Project: Multiplayer Networking in a Modern Game Engine

Type of Project: Application

ELABORATION

As multiplayer networking is ever more prevalent in the modern games market, this project aims to explore existing multiplayer networking techniques, and evaluate their merits and drawbacks to attempt to find a good solution to networking a modern online game.

The project will explore multiplayer networking techniques in a modern game engine. Its focus will be comparing two different networking technologies – TCP/IP and UDP/IP, and discuss the viability of each in terms of networking games. It will explore topics such as packet loss, limitations on clients such as bandwidth constraints and the intervention of technology such as firewalls.

TCP/IP and UDP/IP are two of the most common networking technologies in use today. TCP provides reliable delivery, and UDP provides unreliable delivery. The former operates at the expense of efficiency and speed, and the latter operates at the expense of reliable delivery. UDP is generally used for multiplayer networking, and this project will explore TCP's viability for that application.

Computer Networks is a course that covers a wide range of topics underneath the Computer Science umbrella, including using hardware and software, writing software and designing computer networks. This project complements those subjects, as it collates hardware, software and design knowledge.

PROJECT ETHICS

I will need to take into consideration my research subjects, and make sure that I take care when communicating with them to make sure I do so in a polite manner, and don't antagonise them.

For instance, I will not send multiple unsolicited emails, as this may alienate my subjects, and cause undue burden on the email services that I use for my work.

Any interviews performed with research subjects will be performed with their prior consent, so a time suitable to all parties can be decided upon.

When sending files and acquiring files from the internet, I will also make use of a virus scanner to ensure that I do not accidentally distribute viruses or malware, as this may result in data loss or hardware damage.

Project specification, page 1

PROJECT OBJECTIVES

The student is required to:

- Research existing networking, software and development methods
- Compare each method and tool, evaluating drawbacks and merits
- Decide which tool to use, and how to use it
- Conduct interviews with knowledgeable people
- Create an engine design based on the existing technologies
- Implement, test and document a multiplayer game engine concept
- Consider different user limitations throughout, such as packet loss and latency
- Evaluate quality and usefulness of the application
- Conclusions, recommendations and further work
- Critically evaluate the project after its completion

PROJECT DELIVERABLE

The deliverable for this project will be a prototype multiplayer networked game engine that uses modern networking techniques.

TASK PLAN

Task Name	Start	Finish
Investigate Existing Technologies	Mon 04/11/13	Fri 15/11/13
Investigate existing network protocols	Mon 04/11/13	Fri 15/11/13
Investigate existing game engines	Mon 04/11/13	Fri 15/11/13
Investigate platforms	Mon 04/11/13	Fri 15/11/13
Investigate work of a similar capacity	Mon 04/11/13	Fri 15/11/13
Design and Develop and Engine	Mon 18/11/13	Fri 07/02/14
Design an engine	Mon 18/11/13	Fri 29/11/13
Construct class diagrams	Mon 02/12/13	Fri 13/12/13
Add unit tests	Mon 16/12/13	Fri 17/01/14
Begin barebones code structure	Mon 16/12/13	Fri 27/12/13
Flesh out code from structure	Mon 30/12/13	Fri 10/01/14
Add basic renderer	Mon 13/01/14	Fri 24/01/14
Test prototype and fix bugs	Mon 27/01/14	Fri 07/02/14
Evaluate Application	Mon 10/02/14	Fri 28/02/14
Evaluate quality	Mon 10/02/14	Fri 28/02/14
Evaluate usefulness	Mon 10/02/14	Fri 28/02/14
Evaluate objectives have been met	Mon 10/02/14	Fri 28/02/14
Conclusions, recommendations and further work	Mon 03/03/14	Fri 14/03/14
Critically Evaluate Project	Mon 17/03/14	Fri 28/03/14

Project specification, page 2

ETHICS CHECKLIST

HUMAN PARTICIPANTS

Question	***Yes/No***		
1.	Does the project involve human participants? This includes surveys, questionnaires, observing behaviour, testing etc. *If YES, then please answer questions 2 to 5. If NO, please go to question 6.*	Yes	
2. *Note*	Will any of the participants be vulnerable? *'Vulnerable' means those who may not fully understand the research, or the consequences of taking part. They include:* • *children (i.e. under 18)* • *people with learning disabilities* • *people with physical disabilities (visible or not)* • *people who may be limited by age or illness* *If YES, then please answer question 2a-2b If NO, please go to question 3*	No	
	2a	Will you ever be alone (i.e. not overseen) with any vulnerable participants during the course of the research? *If YES, then please answer question 2b If NO, please go to question 2c.*	-

Project specification, page 3

	2b *Note*	If you will be alone (i.e. not overseen) with any vulnerable participants during the course of the research, do you need to apply for a Criminal Records Bureau (CRB) check (Standard or Enhanced)? *If you need a CRB check, you may be liable for the cost of the application. More details regarding CRB checks can be found at http://www.homeoffice.gov.uk/agencies-public-bodies/crb/*	-
	2c *Note*	If you will NOT be alone (i.e. you will be overseen) with any vulnerable participants during the course of the research, does your overseer have a CRB Check? *If your overseer does NOT have a CRB check, you may need to apply for a CRB check yourself. You may be liable for the cost of the CRB application. More details regarding CRB checks can be found at http://www.homeoffice.gov.uk/agencies-public-bodies/crb/*	-
3. *Note*	Will any participants be at risk of any physical or emotional harm by taking part in your project? *Harm may be caused by:* • *distressing or intrusive interview questions* • *uncomfortable procedures involving the participant,* • *invasion of privacy,* • *topics relating to highly personal information,* • *topics relating to illegal activity, etc.*	No	

Project specification, page 4

4.	Will anyone be taking part without giving their informed consent? (e.g. research involving covert study, coercion of subjects, or where subjects have not fully understood the research etc.)	No
5.	Will any part of the project allow the identification of any individual who has not given their express consent to be identified?	No
Note	*If you answered YES to any of questions 2 – 5, then you MUST address these ethical issues in your Project Specification, ensure that you take all reasonable steps to avoid/mitigate these issues during the execution of your project, and explain your actions in your Critical Reflection.*	

OTHER PARTICIPANTS

Question	***Yes/No***	
6.	Does the project involve the use of live animals?	No
Note	*If you answered YES to question 6, then the project proposal must be submitted to the FREC for approval unless it falls into a category/ programme of research that has already received category approval.*	
7. *Note*	Does the project involve the NHS? *For NHS research, this includes:* • *any service evaluation work* • *work concerning NHS patients (tissues, organs, personal information or data)* • *NHS staff, volunteers, or carers* • *NHS premises or facilities*	No
	If you answered YES to question 7, then your project proposal MUST be submitted Project	

Project specification, page 5

	Module Ethics Committee for review, and may be referred to the Faculty Research Ethics Committee and/or to an NHS Research Ethics Committee. For further details on NHS National Research Ethics Service, please refer to http://www.nres.nhs.uk/applications/faq/before-applying/#FAQsBeforeApplyingStudent	
8.	Does the project require approval from any other external ethics committee?	No
Note	*If you answered YES to question 8, then the project proposal must be submitted to the relevant external body. For further advice, please contact the Faculty Research Ethics Committee*	

ORGANISATIONS

Question	***Yes/No***	
9.	Will the project involve working with or within an organisation? *'Organisation' includes (but is not limited to):* • *business* • *charity* • *museum* • *government department* • *international agency* • *sports/social club* • *volunteer organisation*	No
10. *Note*	If you answered YES to question 9, do you have granted access (permission) to conduct the project? *If YES, please show evidence to your supervisor and include this evidence in the Appendix of your Final Report.*	-
11.	If you answered NO to question 10, is it because: A. you have not yet asked B. you have asked and not yet received and answer C. you have asked and been refused access.	-
Note	*You will only be able to start this aspect of the project when you have been granted*	

Project specification, page 6

Question	***Yes/No***	
	access/permission.	
12. *Note*	Will covert research be part of the project? *'Covert research' refers to research that is conducted without the knowledge of participants.* *If you answered YES to question 12, then your project proposal MUST be submitted Project Module Ethics Committee for review, and may be referred to the Faculty Research Ethics Committee*	No

Project specification, page 7

PRODUCTS AND ARTEFACTS

Question	***Yes/No***	
13 *Note*	Will the project involve using (e.g. citing / quoting / copying) copyrighted materials? *Copyrighted materials includes (but is not limited to):* • *books / e-books* • *journals* • *websites* • *newspaper/magazine articles* • *films / broadcasts* • *photographs, artworks, images, diagrams* • *designs, products* • *computer programmes, code, databases* • *networks* • *processes* *If YES, please go to question 14* *If NO, please read the declaration at the end of the checklist*	Yes
14. *Note*	Are the copyrighted materials you intend to use (citing / quoting / copying) in the public domain? *'In the public domain' does not mean the same thing as 'publicly accessible'.* • *Information which is 'in the public domain' is no longer protected by copyright (i.e. copyright has either expired or been waived) and can be used without permission.* • *Information which is 'publicly accessible' (e.g. TV broadcasts, websites, artworks, newspapers) is available for anyone to consult/view. It is still protected by copyright even if there is no copyright notice. In UK law, copyright protection is automatic and does*	No

Project specification, page 8

Question	Yes/No	
	not require a copyright statement, although it is always good practice to provide one. It is necessary to check the terms and conditions of use to find out exactly how the material may be reused etc. *If YES, please read the declaration at the end of the checklist.* *If NO, please go to question 15.*	

Project specification, page 9

Question	***Yes/No***		
15. *Note*	Will the project involve copying/reproducing copyrighted materials? *'Copying/reproducing' includes making physical copies (i.e. photocopies) and electronic copies (i.e. cutting-and-pasting, scanning, saving as separate files). It does not include citations and quotes with appropriate references.* *If YES, please go to question 15a.* *If NO, please go to question 16.*	No	
	15a *Note*	Will you be copying more than 5% (or one chapter) of an individual source? *You are allowed to copy/reproduce up to 5% of a source (or one chapter) for your own personal research usage without explicit permission. Please see "Copyright - guidance for SHU Staff and Students" for further information.*	-
16. *Note*	Do you have permission to use the copyrighted materials under the "Exam Defence"? *If you are copying less than 5% (or one chapter) or citing/quoting the copyrighted material, you have permission to use the copyrighted material under the "Exam Defence". Please see "Copyright - guidance for SHU Staff and Students" for further information.* *If YES, please read the declaration at the end of the checklist.* *If NO, please go to question 17.*	Yes	

Project specification, page 10

Question	***Yes/No***	
17.	Do you have explicit permission to use the copyrighted materials? *If YES, please show any explicit evidence to your supervisor and include in the Appendix of your Final Report. Please read the declaration at the end of the checklist.* *If NO, please go to question 18.*	-
18.	If you do not have explicit permission, is it because: A. you have not yet asked permission B. you have asked and not yet received and answer C. you have asked and been refused access.	-
Note	*You will not be allowed to use the copyrighted material until you have been granted permission to use it.*	

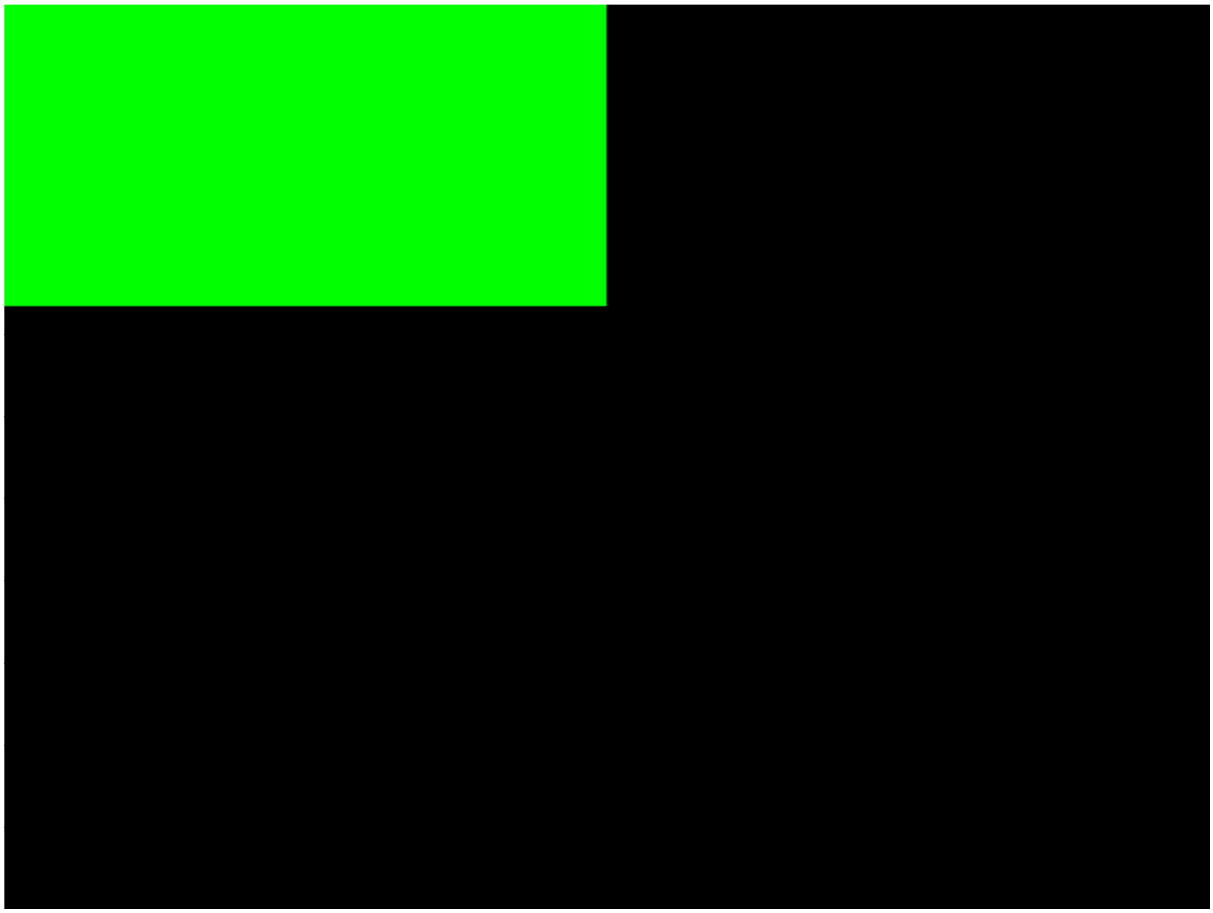
ADHERENCE TO SHU POLICY & PROCEDURES

Declaration	
	I can confirm that: • I have read the Sheffield Hallam University Research Ethics Policy and Procedures (available at http://www.shu.ac.uk/research/downloads/ethicspolicy2004.pdf)) • I agree to abide by its principles.

Project specification, page 11

Appendix 2 Project Schedule

Appendix 3 Test Code Reference Rendering



Appendix 4 Pure OpenGL Implementation of Reference Rendering

```
#include <gl\freeglut.h>

void display()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    const float rectangleWidth = 400.0f;
    const float rectangleHeight = 200.0f;

    glBegin(GL_QUADS);
        glColor3f(0.0f, 1.0f, 0.0f);
        glVertex2f(0.0f, 0.0f);
        glVertex2f(rectangleWidth, 0.0f);
        glVertex2f(rectangleWidth, rectangleHeight);
        glVertex2f(0.0f, rectangleHeight);
    glEnd();

    glFlush();
    glutPostRedisplay();
}

int main(int argc, char** argv)
{
    const int width = 800;
    const int height = 600;

    glutInit(&argc, argv);
    glutInitWindowSize(width, height);
    glutCreateWindow("OpenGLTest");

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0f, width, height, 1.0f, -1.0f, 1.0f);

    glutDisplayFunc(display);
    glutMainLoop();

    return 0;
}
```

Appendix 5 SFML Implementation of Reference Rendering

```
#include <SFML\Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(800, 600), "SFMLTest");

    sf::RectangleShape shape(sf::Vector2f(400, 200));
    shape.setFillColor(sf::Color::Green);

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        window.clear();
        window.draw(shape);
        window.display();
    }
}
```

Appendix 6 Allegro Implementation of Reference Rendering

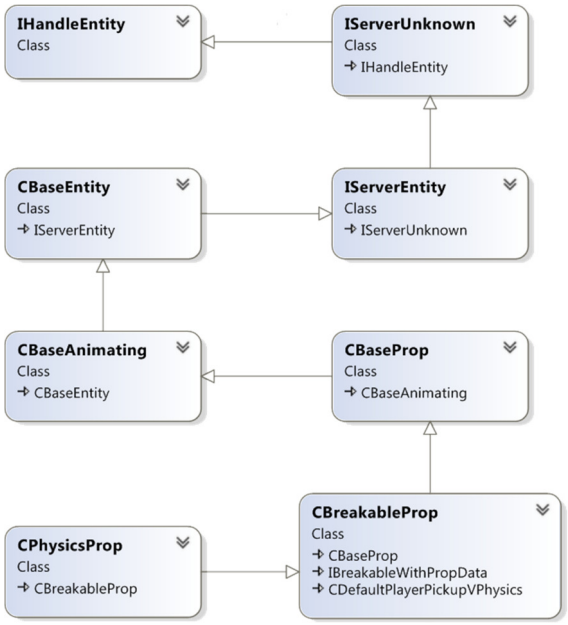
```
#include <allegro5\allegro.h>
#include <allegro5\allegro_primitives.h>

int main()
{
    int width = 800;
    int height = 600;

    ALLEGRO_DISPLAY *display = NULL;

    al_init();

    display = al_create_display(width, height);

    al_init_primitives_addon();

    ALLEGRO_EVENT_QUEUE *EventQueue;
    ALLEGRO_EVENT Event;
    EventQueue = al_create_event_queue();
    al_register_event_source(EventQueue, al_get_display_event_source(display));

    do
    {
        al_draw_filled_rectangle(0, 0, 400, 200, al_map_rgb(0, 255, 0));
        al_flip_display();
        al_wait_for_event(EventQueue, &Event);
    }
    while(Event.type != ALLEGRO_EVENT_DISPLAY_CLOSE);

    al_destroy_display(display);

    return 0;
}
```

Appendix 7 Qt Implementation of Reference Rendering

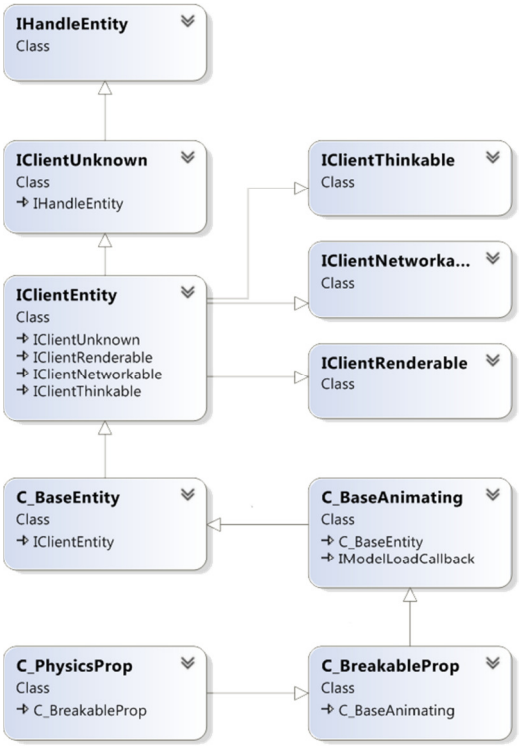
```
#ifndef PAINTWIDGET_H
#define PAINTWIDGET_H

#include "qwidget.h"
#include "qpainter.h"

class PaintWidget : public QWidget
{
public:
    PaintWidget(QWidget *centralWidget) : QWidget(centralWidget)
    {
    }

    void paintEvent(QPaintEvent *)
    {
        QPainter painter(this);
        painter.setRenderHint(QPainter::Antialiasing);
        QRect rect = QRect(0, 0, 400, 200);
        painter.fillRect(rect, QBrush(QColor(0, 255, 0, 255)));
    }
};

#endif // PAINTWIDGET_H
```

Appendix 8 Source Engine Server and Client Class Diagram

Server



Client



Appendix 9 Source Engine Server Class for a Dynamic Light

```
20  class CDynamicLight : public CBaseEntity
21  {
22  public:
23      DECLARE_CLASS( CDynamicLight, CBaseEntity );
24
25      void Spawn( void );
26      void DynamicLightThink( void );
27      bool KeyValue( const char *szKeyName, const char *szValue );
28
29      DECLARE_SERVERCLASS();
30      DECLARE_DATADESC();
31
32      // Turn on and off the light
33      void InputTurnOn( inputdata_t &inputdata );
34      void InputTurnOff( inputdata_t &inputdata );
35      void InputToggle( inputdata_t &inputdata );
36
37  public:
38      unsigned char m_ActualFlags;
39      CNetworkVar( unsigned char, m_Flags );
40      CNetworkVar( unsigned char, m_LightStyle );
41      bool    m_On;
42      CNetworkVar( float, m_Radius );
43      CNetworkVar( int, m_Exponent );
44      CNetworkVar( float, m_InnerAngle );
45      CNetworkVar( float, m_OuterAngle );
46      CNetworkVar( float, m_SpotRadius );
47  };
```

Appendix 10 Source Engine Client Class for a Dynamic Light

```
33 class C_DynamicLight : public C_BaseEntity
34 {
35 public:
36     DECLARE_CLASS( C_DynamicLight, C_BaseEntity );
37     DECLARE_CLIENTCLASS();
38
39     C_DynamicLight();
40
41 public:
42     void    OnDataChanged(DataUpdateType_t updateType);
43     bool    ShouldDraw();
44     void    ClientThink( void );
45     void    Release( void );
46
47     unsigned char   m_Flags;
48     unsigned char   m_LightStyle;
49
50     float   m_Radius;
51     int     m_Exponent;
52     float   m_InnerAngle;
53     float   m_OuterAngle;
54     float   m_SpotRadius;
55
56 private:
57     dlight_t*   m_pDynamicLight;
58     dlight_t*   m_pSpotlightEnd;
59
60
61     inline bool ShouldBeElight()
62         { return (m_Flags & DLIGHT_NO_WORLD_ILLUMINATION); }
63 };
```

Appendix 11 Interviewee Background Information and Consent Form

Interviewee Background Information and Consent Form

Project Background

The aim of this project is to design a prototype 2D multiplayer networked engine, comparing the usage of TCP/IP with UDP/IP. It will include physics simulation and packet loss compensation, and will be designed using research gained from existing projects, other research projects and research performed for this project.

Interview Purpose

The purpose of this interview is to research information about how to design the engine, with interview candidates with varied experience in associated areas. Interview results may be used to design the engine codebase, but will remain completely anonymous.

Consent

In order to use any information provided, consent must be given to certify that the above information has been read and understood.

By signing the below, the subject agrees that answers provided may be used as research to design the engine. As mentioned above, candidates will remain completely anonymous, and possess the right to withdraw from the project at any time.

Interviewee Consent

I agree that I have read and fully understand the information stated above.

Full Name of Interviewee	Signature of Interviewee	Sign Date (DD/MM/YY)

Interviewer Consent

I agree to abide by the guidelines stated above.

Full Name of Interviewer	Signature of Interviewer	Sign Date (DD/MM/YY)

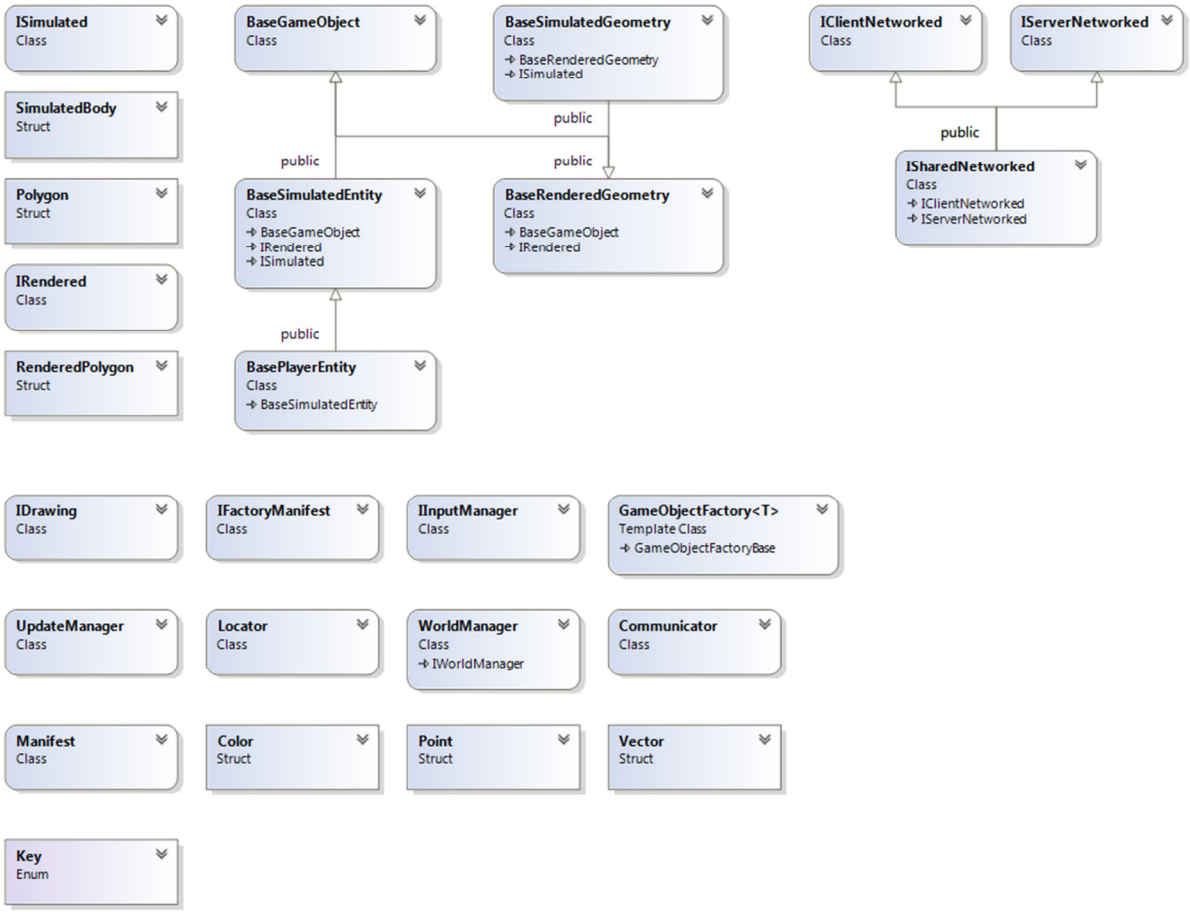
Appendix 12 Interview Results

Q1. When structuring a game engine, different components of that engine have to interact, and most components will have dependencies on other components. Which design pattern best suits the management of this?
I don't think there is a specific best design pattern as it depends on the context, which changes depending on the developers, language and intended purpose of the engine (it's rare that engines are really totally generic). The most generic design I've seen is Unity which uses the entity-component model. I use singletons quite a lot, but they are nearly as weak as global variables (which are terrible). Writing generic code is hard and takes 10x longer than just building a game around a specific problem. The flash approach (observer pattern) of triggering event notifications which other objects can register to listen to also has a lot of advantages. Another approach to build objects behind interfaces (pure virtual abstract base classes which can be used with multiple inheritance without danger) that control all interaction, which also has advantages/disadvantages.
Design patterns are general solutions to common problems and so it would be difficult to pick one that would solve all your issues. Dependency injection would reduce coupling and would lend well to unit testing. A singleton would be one approach for say a ResourceManager or Scene Graph type object, but you could also explore the service locator pattern which admittedly also has its problems. Arguments against DI, singletons and global variables are that by using any of them you're breaking encapsulation. DI allows an external component to inject its behaviour thus modifying the internal workings of that class, global variables obviously allow other code to access/modify their values which can introduce bugs/unwanted behaviour. Singletons are generally difficult to Unit test and break the 'S' (single responsibility principle) of the SOLID principles (considered best practice) by having mixed responsibilities and therefore reducing cohesion. You would need to consider the strengths/weaknesses of each and make a decision based on risk/reward for what you need your code to do. I'd encourage basic unit/acceptance testing to ensure your code is scalable and potential bugs are discovered earlier in the development cycle.
You'd probably do well to look at the service locator pattern, but roll on your own instead of using an off the shelf dependency injection container which will be slow. You register stuff at start up with a singleton service locator and create game components which have (generally)... Update, Draw, IServiceLocator, create an interface for this. Then in your game loop you can simply call update on the application idle event and call draw every frame to decouple physics and drawing. Quite often you can hit 60fps but be updating your physics 500 times a second. Your components then can use "ServiceLocator GetService IMyGraphicsDeviceAbstraction" or whatever... IMyResourceManager, IMyStorageProvider and stuff.
Dependency injection is a great idea, which adds great flexibility. Structuring it in a way that allows easy debugging is always a good idea. There's a fine mix between different methods, especially with a game engine, which incorporates many different things, such as audio, networking, rendering, etc.
Using singletons and globals may work fine initially but if you create and use them all over the place it might be useful to consider using static classes or service locators. Another danger of using them is that you can easily start coupling code. Where you suddenly are playing audio when a physics object is colliding, *IN* the actual physics code. That wouldn't be good. I'm not a design pattern genius though.
Q2. Do you think C++ is a good language decision for a prototype 2D engine with

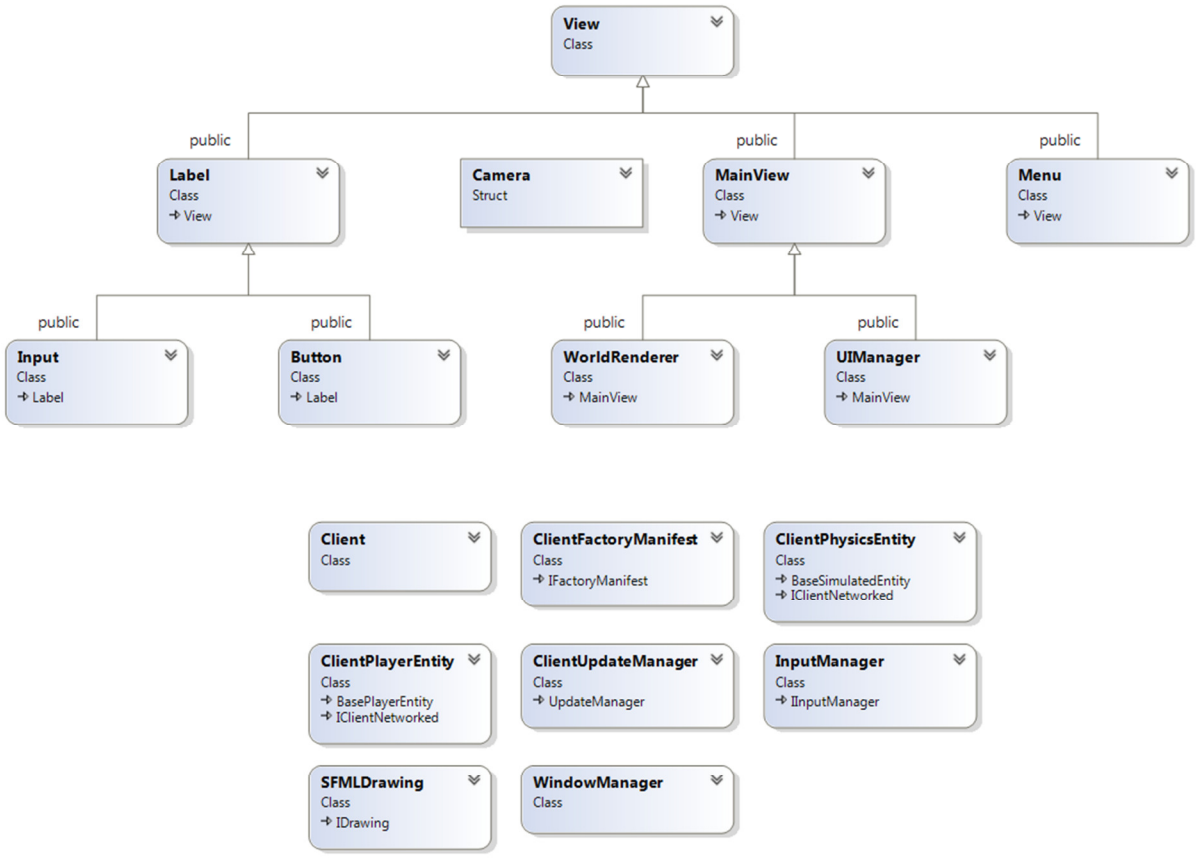
networking?
C++ is designed for speed and size (small) of executable which is why it's used so much in game development. Whether it's a good choice for a prototype is utterly dependant on the goals of the prototype. A prototype is an experiment to answer a question, knowing the question you can work out the quickest way to a possible answer. If the question is "can I build a 2D networked engine using DX11" then if I was a C++ guru with knowledge of sockets and threads then C++ would be an obvious choice.
I think C++ would not be my first language choice but a lot of that decision would come down to the developer. Microsoft and Java have strong networking libraries; Microsoft also has the XNA framework which contains all the components needed to get a small prototype up and get running without the developer needing to wrap all the windows programming boiler plate code.
I'd have probably used "XNA", "SharpDX", or "Managed DX" in C# to allow you to concentrate on the core of your thesis and not so much dicking around initializing devices, etc. That said, my 3d stuff for University was C++... So whatever you're comfortable with.
C++ is a great language to go with. It can be quite portable, it has many open source libraries that are lightweight and easy to implement. It would also give you greater performance over something like Java, C# or similar languages.
With the use of some helpful libraries using C++ should allow you to write a 2D engine at a decent speed. When it comes to prototyping your dependency injection plans will likely make things a lot easier. The difficulty of prototyping code changes will likely depend on the base structure of the engine.
Q3. How large should individual network update packets be, and how frequently should they be exchanged?
It depends how you plan to deploy games, is the network peer-peer, is it only for use on a LAN or is it over the net, are you requiring a server? Broadband is very unreliable, and the net is optimised for web pages. Latency could be as bad as 250-500mSecs and it has to be less than 100mSecs to feel like instant – so is this engine for reflex based games or turn based ones? A web page is ~50k, possibly delivered in 1.5-3k packets. So in game terms that sounds like quite a bit of data, delivered not that often.
There would be many factors to consider, such as protocol volume of data and network structure. I would suggest (if using TCP) to make the frequency and size configurable as the situation may vary.
That's a tough question. I'd say they need to be kept under the size of an MTU so the packets aren't fragmented and need reassembling at the destination (which is slow / latent) I think that's around 1498 bytes as standard for DirectX. Also, how frequent... Hmm... Tricky, depends on use... For massively scalable games (MMO) as little traffic as possible, for other stuffs (FPS's) then very frequent, it all depends on how well you can use prediction algorithms to 'smooth' between packets. A driving game for example would be much easier to predict the behaviour of a network players car, given the forces operating, the last know controller inputs, etc., and the shape of the track. It DOES mean however, that you may need to send more state in your packets (think thumb stick positions, and current amount of steer), not just a position and a heading.
It entirely depends on your application. E.g. Valve's Source Engine sends updates around 60 times per second... When it's something like an FPS game, once you get around 20 things really start to suffer. Taking about MTUs; I think the minimum MTU is >= 500. The usual average is roughly 1500, minus the header you have something around 1470 MTU. But, the answer could be argued by looking at the MTU of the network... So ideal packet size prevents packets from being split up for transmission, etc.
For games they should be exchanged at a regular interval. At least thirty per second for competitive games (preferably more). I'm not so sure about the size of the packets. Keep

them as small as possible.
Q4. With regards to a "main loop", should threading be used to separate rendering and simulation? When should network packets be processed?
Rendering and game logic should not be in separate threads in a game client, in a server there is no rendering so it's different. Rendering and logic can still run at different rates even if they aren't in different threads. What does need to be in a different thread is code dealing with networking, whether it's a game client processing messages or a server (possibly on one of the players' computers).
Threading would be a good way to handle not just networking, but possibly other resource heavy tasks as well. In terms of when the packets should be processed, again it would depend on what you were using the data for.
Given that modern CPU's have many cores, you'd probably want to use threads for Drawing, Updates, Network, Sound as a minimum. It involves a bit of jiggery pokery to prevent blocking, for example, it's probably better to always lock the vertex buffers on the drawing thread, but just do a try lock on the update thread and skip the update if it can't lock. You can also do clever stuff with maintaining separate lists as = is thread safe. I wrote a blog post about thread safety, it's on [sequensis.azurewebsites.net]. Note how I copy the list, manipulate the copy, and then reassign the list at the end. This involves no locking but is perfectly thread safe the C# standard guarantees the equals operator is thread safe, same for C++ i believe.
Many engines do it differently. In my 'experience' "main loop" should run 'update' things, and then you have a thread for physics and a thread for rendering.
It would be useful to separate them but if you're using a third party simulation library then there is a good chance it does the threading for you. Packets should probably be processed after events have fired. That way you can use the packet data for prediction and interpolation of game objects when the engine is executing logic functions.

Appendix 13 Shared Code Class Diagram



Appendix 14 Client Library Class Diagram



Appendix 15 Server Library Class Diagram

Server Library Class Diagram

Appendix 16 Manifest Class Header

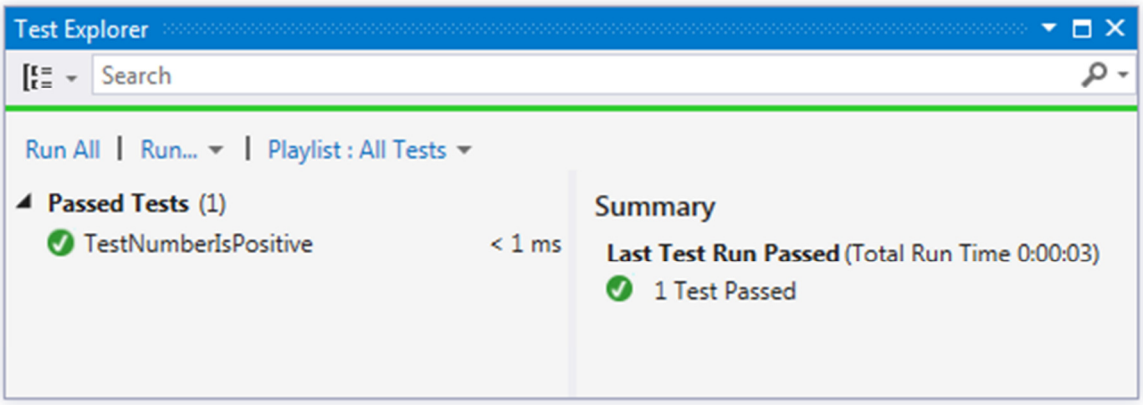
```
class Manifest
{
public:
    virtual bool GetBool(std::string szKey, bool bDefault = false);
    virtual int GetInt(std::string szKey, int iDefault = 0);
    virtual float GetFloat(std::string szKey, float flDefault = 0.0f);
    virtual std::string GetString(std::string szKey, std::string szDefault = std::string());
    virtual Manifest GetManifest(std::string szKey);
    virtual void SetBool(std::string szKey, bool bValue);
    virtual void SetInt(std::string szKey, int iValue);
    virtual void SetFloat(std::string szKey, float flValue);
    virtual void SetString(std::string szKey, std::string szValue);
    virtual void SetManifest(std::string szKey, Manifest oManifest);
    virtual void SetManifestList(std::string szKey, std::vector<Manifest> oManifestList);
    virtual void WriteManifest(std::string szFilename = std::string());
};
```

Appendix 17 World Renderer Draw Method Implementation

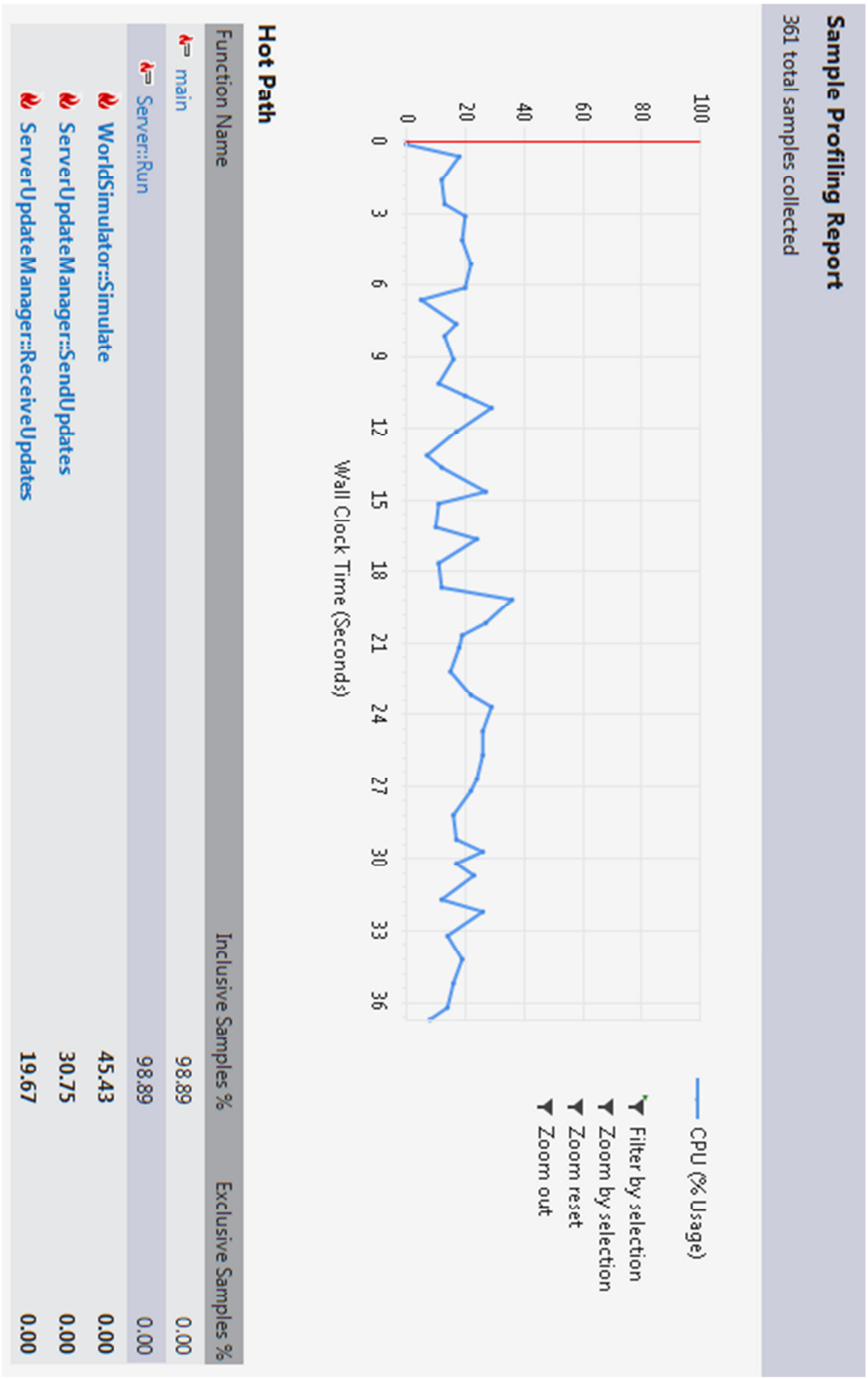
```
60  void WorldRenderer::Draw()
61  {
62      // Acquire a list of entities
63      auto oEntities = Locator::WorldManager()->GetGameObjects();
64
65      // Iterate the list
66      for (auto pEntity : oEntities)
67      {
68          // If the entity is deleted, skip it
69          if (pEntity->deleted)
70              continue;
71
72          // Cast the BaseGameObject to an IRendererd
73          auto pRenderable = dynamic_cast<IRendered*>(pEntity);
74
75          // Check that the object implements
76          // IRendered (if it doesn't, don't render it)
77          if (pRenderable)
78          {
79              // Check if the current entity is a player entity
80              auto pPlayer = (ClientPlayerEntity*)pEntity;
81
82              // If it is, and this client owns the player
83              if (pPlayer && Locator::GameState()->UpdateClientId() == pPlayer->playerId)
84              {
85                  // Smooth the camera position - use 90% of the old, 10% of the entity position
86                  // Given time, this is smoothed out. TODO: make this use delta time
87                  m_oCamera.position = (m_oCamera.position * .9f) + (pPlayer->position * .1f);
88              }
89
90              // Draw the renderable
91              DrawRenderable(pEntity, pRenderable);
92          }
93      }
94
95      // Check if the engine is in debug mode
96      if (Locator::GameState()->Settings()->GetBool("debug"))
97      {
98          // Loop Entities
99          for (auto pEntity : oEntities)
100         {
101             // If the entity is deleted, skip
102             if (pEntity->deleted)
103                 continue;
104
105             // Draw debug text over the entity
106             DrawDebugText(pEntity);
107
108             // Case the entity to a simulated entity
109             auto pSimulated = dynamic_cast<ISimulated*>(pEntity);
110
111             // If it is a simulated entity, draw its collision hull
112             if (pSimulated)
113                 DrawSimulated(pEntity, pSimulated);
114         }
115     }
116 }
```

Appendix 18 Example C++ Unit Test and Visual Studio Result

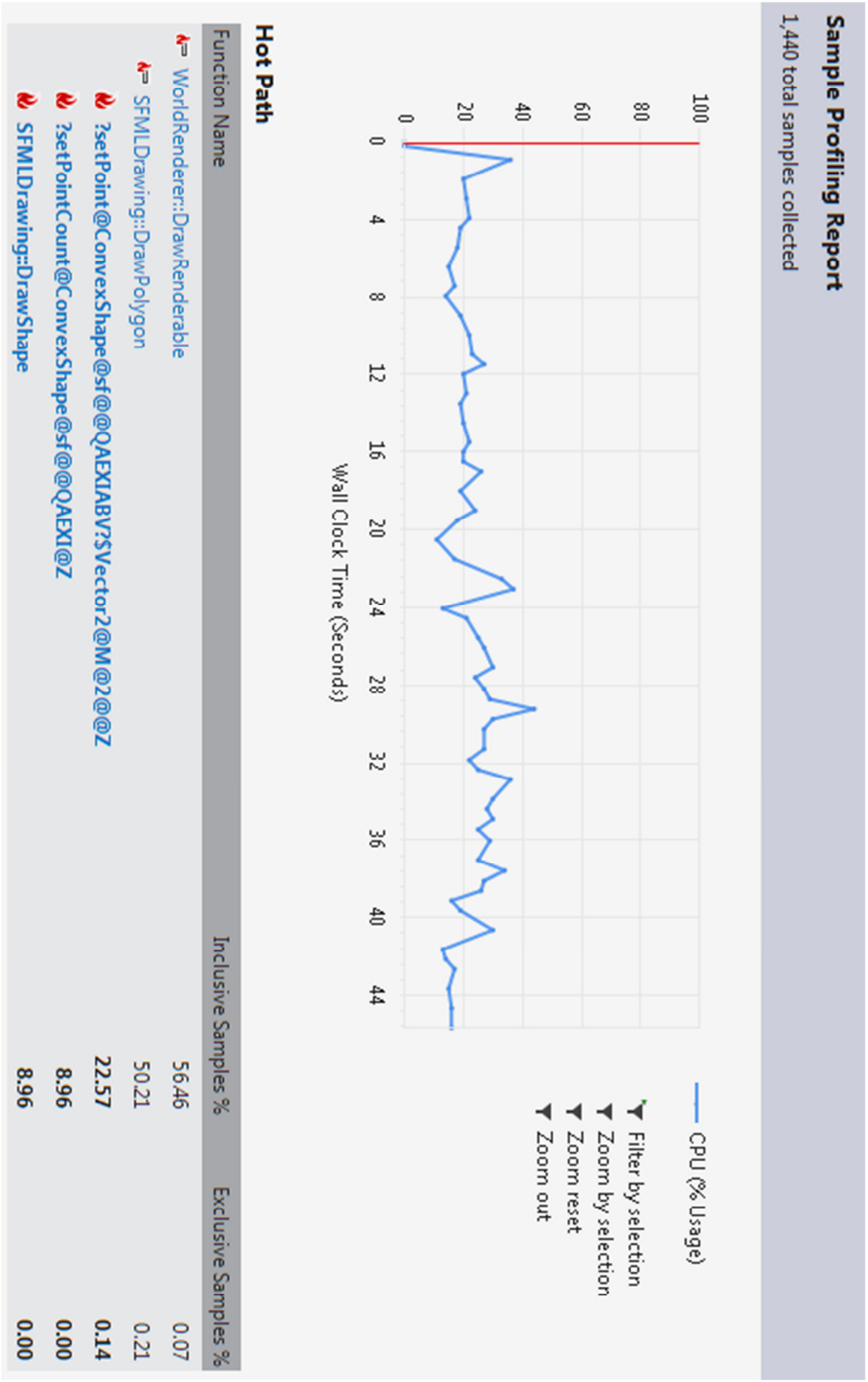
```cpp
#include "stdafx.h"
#include "CppUnitTest.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

class ClassToTest
{
public:
    int GetPositiveNumber() { return 42; }
};

TEST_CLASS(UnitTestClass)
{
public:
    TEST_METHOD(TestNumberIsPositive)
    {
        // Create the class we're testing
        auto pTestInstance = new ClassToTest();

        // Get the result of a method in the class
        auto iPositiveNumber = pTestInstance->GetPositiveNumber();

        // Assert that the answer is positive
        Assert::IsTrue(iPositiveNumber > 0);
    }
};
```



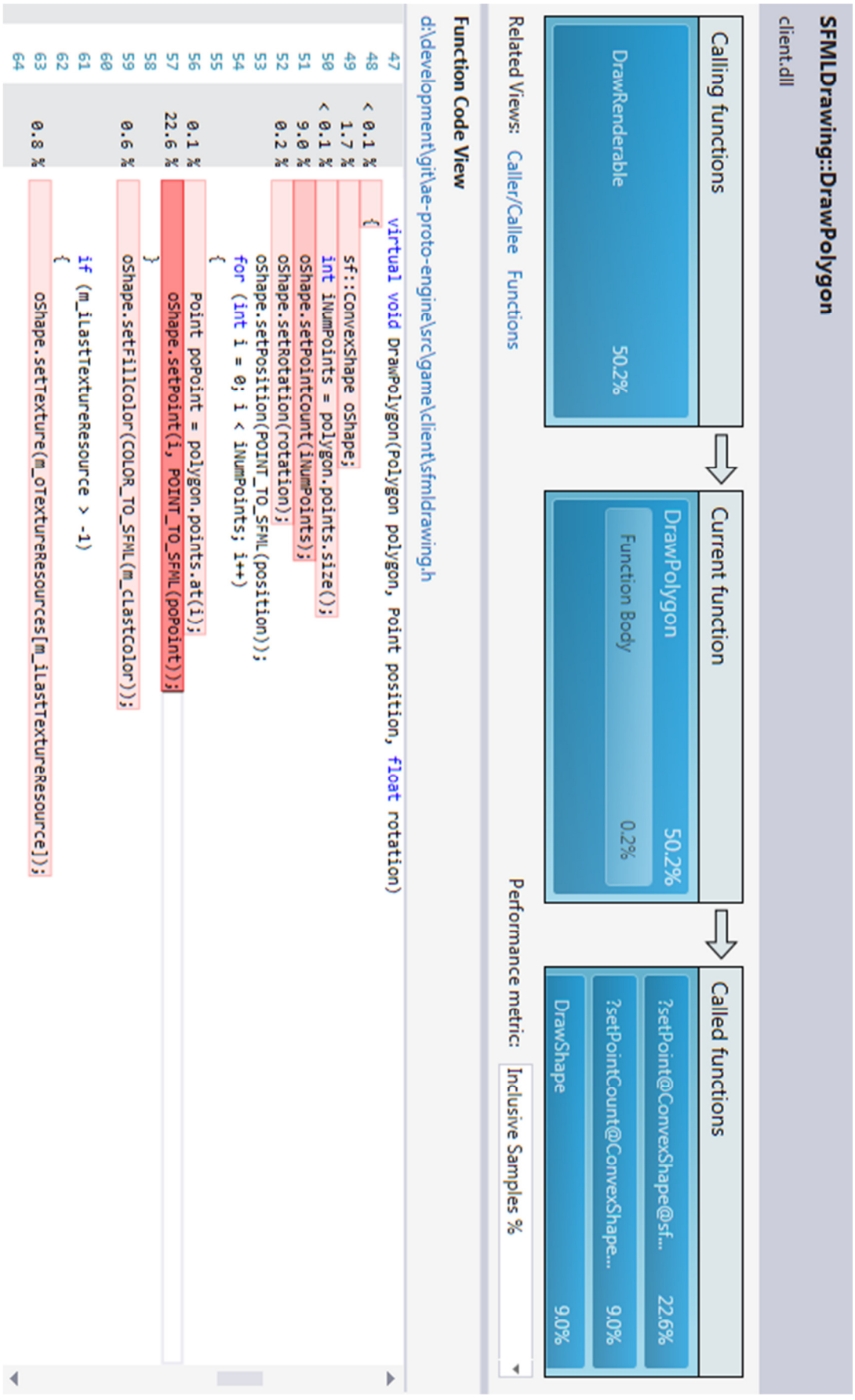
Appendix 19 Performance Analysis on the Server Project



Appendix 20 Performance Analysis on the Client Project



Appendix 21 Further Performance Investigation into Rendering Code



Appendix 22 WireShark Packet Trace Result