DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF LEICESTER

DISSERTATION

Project MOBO

Christopher J. Cola

MAY 2016

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s).

Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s).

I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Christopher J. Cola
Signed: Christopher J. Cola
Date: 09/04/2016

1 Abstract

Online multiplayer video games often require meticulously designed distributed application structure and communications protocols to achieve synchronization of the game state across clients. Nowadays, as developers put increasing emphasis on providing complex network features as part of their software releases, there is a need to correctly design, implement and maintain their distributed system, for both error prevention and performance efficiency.

In this project I aimed to create an online multiplayer game, characteristically similar to open-source game XKobo, a game where players controlling spaceships attempt to shoot and destroy tree-like space stations. The implementation takes the form of a client-server architecture, in which the client is a XNA framework based application, containing the game logic and 2D graphics, and the server is used as a non-authoritative message distributor and maintains an up to date data model representing the game state.

Communications protocols and specific message types ensure synchronization of many aspects of the game state such as the position, health and score of players in game. Stations, which are randomly generated using an algorithm are also synchronized in a way that maintains their state across clients should they get partially destroyed by players, enabling seamless drop-in-drop-out gameplay where new players always receive an up to date version of the game state.

The resultant software, following significant test-driven development, achieves a stable and synchronous multiplayer experience, and has many features one might expect of a modern day title such a polished graphical user interface, smooth performance and visual effects. The project provided valuable insight into the complex programming challenges encountered in game development, as well as the difficulty in maintaining a state of synchronization between clients that are concurrently interacting with the game state.

2 Table of Contents

3 Introduction

3.1 Background

In 1997 Akira Higuchi released a free and open-source video game called XKobo for the UNIX operating system, written in the C programming language. XKobo is a single player, arcade style space shooter in which the player assumes control of a space ship in battle with enemy space stations. The space stations consist of a core which branches out in a tree-like structure from the center and is connected to further nodes which act as turrets, firing projectiles at the player if they get within range. Players are able to destroy turret nodes by shooting at them however the only way to destroy a station outright is to destroy the core. The objective of the game is to destroy all the space stations in the level, at which point the player proceeds to the next. Originally utilizing the X Window System to draw graphics on the screen and obtain input from the keyboard, XKobo has experienced numerous revisions over the years since its initial release and has been ported to modern operating systems by passionate fans.

As is conventional for the information technology industry in general, much has progressed in the almost two decades since XKobo was released, in terms of both the technology that is now available for software developers to assist in video games development and the user's expectation of the software, specifically the features and user experience it should provide. When XKobo was originally released, the video games industry was on the advent of popular and commercially available multiplayer games playable over an internet-connected network. DOOM, released in 1993 by id Software, was not the first example of video game with networked multiplayer possible over the internet, but it was successful in popularizing the concept that users could play in the same synchronized game instance over the internet. Nowadays developers are putting increasing emphasis on providing network gameplay as part of their software releases, with many studios focusing on exclusively online games. As video games become ever more complex in functionality the need to correctly design, implement and maintain them has never been more apparent.

3.2 Aims and Objectives

The overall aim of the project was to produce a game characteristically similar to XKobo but with a focus on providing multiplayer online gameplay as a feature. The game itself, which ultimately became the client half in a client-server modelled system, needed to utilize a 2D graphics engine or framework to display graphics representing game mechanics to the user, such as players, projectiles and enemies. The chosen framework was also required to display an intuitive graphical user interface that displays relevant information to the user and enables them to configure their gameplay experience. Although not specifically stated in the project description, another aim was to ensure the user experience of the client was high with an attention to aesthetically pleasing and consistent graphics, graphical user interface elements and visual effects seen during gameplay.

On the server side, the key aim was to enable synchronization of game state between connected clients by developing client and server specific network messaging protocols. Synchronization in the context of networked games refers to each client having an as up to date as possible view of the game state as possible, provided by the server. This is achieved by the server sending data regularly to connected clients representing its current state while receiving player-specific data from clients and updating its own state to include the actions of the players. For example, if a player in an online session shoots an enemy and it is destroyed, all remaining connected clients should know when and where the specific player fired the projectile, that the projectile hit the enemy and the enemy was destroyed as a result. Synchronization therefore depends on the messages (or packets) that are transmitted between client and server, what the data they contain pertains to, and the way in which these messages are sorted and the data parsed.

Research would form a large part of the project initially as there inherently many ways one could go about producing a client-server system, and many technologies available that enable this in varying degrees of flexibility and methods. On top of this, freedom was given in the project proposal of which programming language to use to produce the client and server and since 2D graphics APIs are usually bound to specific programming languages (i.e. the Graphics2D API for Java), the technology used to implement the system would be carefully considered. Freedom was also given the game design, whilst the game had to have resemblance to the gameplay of XKobo, the game logic and visual design were open-ended.

Software deliverables were two-fold, to develop a game client utilizing 2D a graphics framework to display the game itself, a graphical user interface, the players, enemy stations and whatever additional elements the game required, and a separate server application that enables and assists a synchronized and robust multiplayer gameplay experience.

4 Requirements specification

4.1 Overview

This section discusses the requirements for the software system and details the reasons for their necessity to provide a more detailed specification of what the project should achieve in comparison the aims and objectives. Throughout the course of the project the requirements were frequently altered as some of the more nebulous aspects of the project became clearer, and changes were made to better reflect the direction of the project.

4.2 Graphical user interface

The client will present a graphical user interface to the user suitable for offering the available options and processing input for the user in the menus and displaying useful information in-game.

The graphical user interface (GUI) is responsible for a large portion of the user's interaction of the software. Upon initial execution of the software, the user will be greeted with a menu screen that contains various options. Among these options at a minimum will be the ability to start a new offline game, join an online networked game, navigate to a settings screen, view a help screen that displays various gameplay specific instructions and hints and an option to exit the game. The GUI should capture mouse input so that the user can click the option they desire, and the cursor should change based on context, i.e. change to a caret for text input and a pointer or hand for an element that can be clicked.

The settings screen should be separate from the menu screen and will provide options for tweaking various settings that the client provides. Players should be able to set their player name, resolution of the game and network specific options such as setting the IP address and port number used to establish a connection to the server for network play. The window of the game should be resizable to accommodate different sized screens if the user desires. The GUI should respond to the change by displaying elements using relative positioning as opposed to absolute, which could lead to UI elements being positioned incorrectly at different resolutions.

In-game the GUI should feature a heads-up display (HUD) that displays game related information to the user during gameplay. This HUD will feature a health bar, a visual indicator of the health of the player's spaceship, a list of the current players in the server and their respective scores, and a message pane. The message pane will display regular updates related to the game state for example, a new player joining or leaving the server, a player dying, or to indicate the arrival of new enemies. This is fairly open-ended and there should be enough messages displayed to the user to relay all necessary information to the player but there should not be so many messages that the player is overwhelmed with text on the screen.

4.3 Capturing player input

While in-game, the client will capture input from the user's mouse and keyboard so that the user is able to move their character, shoot enemies and return to the menu.

The player will use the keyboard arrow keys or the WASD keys (a very common mapping for movement in modern games) to move their character. The controls should be responsive, and should not be overly sensitive or delayed which could harm the user experience. The player character should begin moving in the respective direction that was pressed until the key is released, at which point they slow to a stop. At the same time, the player will be able to aim using the mouse. A crosshair will replace the mouse cursor while in-game and if the left mouse button is clicked the player character will shoot in the direction of crosshair. Pressing the Esc key should return the user to the main menu.

The sprite (the computer graphic that represents the spaceship) should face in the direction of where it is aiming (the position of the mouse) when the ship is stationary. When moving, the sprite should face in direction of travel, a result of the directional key being pressed. Not only does this improve the perceived visual quality of the game, but it also provides feedback on the current direction of the player should the player temporarily forget what key is pressed or the position of their mouse.

4.4 Mini-map

While in-game, client will display a mini-map as part of the graphical user interface so that the user is able to track his/her allies and enemies relative to their current position.

The original XKobo game features a mini-map as part of the GUI. Mini-maps are a common feature of video games that assist players in locating and orientating themselves within the game environment. They usually appear as a miniaturized version of the overall level and have pointers to indicate positions of notable game objects. For this project, the mini-map will display the locations of allies and enemy space stations and thus will behave very similar to conventional radar. To help players distinguish between other players and enemies, other players will appear on the mini-map as blue dots, and enemy space stations will appear red.

The mini-map should update frequently, potentially in real time. As the player moves around the mini-map should update to reflect this since it is used like radar to gauge relative distance between the player and other objects. Whenever new players join they should appear on the radar, and if they leave they should disappear. Similarly, when new stations are spawned, they should appear on the radar and when they are destroyed, removed from it. Since stations are complicated structures as compared to single entities like players, the representation of the stations on the radar should look like miniature versions of their structure, much like in the original XKobo.

4.5 Procedural station generation

While in-game, the client will procedurally create new, unique stations by using an algorithm that randomly generates the structure and shape of a station based on set parameters.

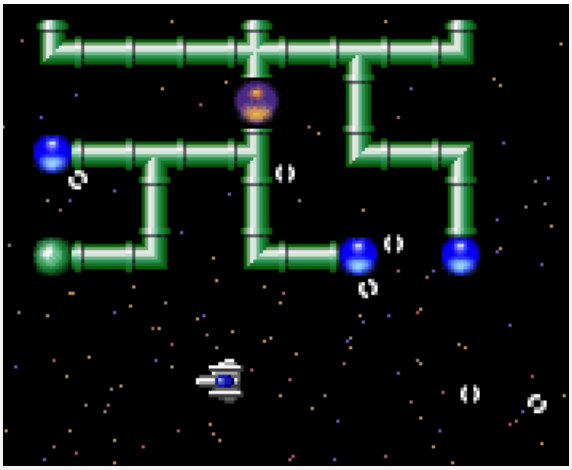

Figure 1 - A screenshot of a station in XKobo.

In XKobo, stations are the main enemies of the game and they vary in structure and size. They are tree-like in structure and extend from a core (which can be thought of as the root of the tree) and branch off in pipes to form turret nodes (the leaf nodes of the tree.) All of the different permutations of structure however were pre-programmed by Higuchi and are tied to the level, for instance if you repeat a level in XKobo, the station will be exactly the same as the station you previously fought. For this project, I sought to improve upon XKobo and introduce a station generator that produces unique stations every time, improving the replayability and reducing repetitiveness. The generation will depend on set parameters, namely a size, and may include additional parameters such as number of branches and number of turret nodes.

It is important to achieve the right balance with random station generation. If the algorithm produces stations that vary highly in complexity then it may produce both stations that are very small and are too easy to destroy and stations that are too large and contain too many station nodes making them overly difficult to destroy. The performance of the station generator is also important. Considering stations will be spawning frequently while the user is playing the game to replace stations the player has destroyed, the algorithm that produces the stations needs to be as fast and resource efficient as possible. If the generator algorithm is performance intensive and requires a lot of computational resources, the game may slow down or altogether stop while the algorithm runs.

The randomly generated station will be displayed using tile graphics for each node to suit the orientation and number of children that a node may have. This was achieved in XKobo by utilizing a sprite sheet with different permutations and orientations of pipe graphics. Turret nodes and the core node should have a distinct appearance different from the pipe nodes so that they can be easily identified.

4.6 Collision detection

The client will use collision detection to check for intersections between entities such as players, projectiles and stations and act accordingly based on the nature of the interaction.

Collision detection is a broad term in computing that refers to the detection of the intersection of entities. The client will need to know when projectiles fired by other players or turret nodes from stations hit the player so that their health can be decreased accordingly. Conversely, clients also need to detect when player projectiles hit station nodes so they can be destroyed and removed from the station. In addition, if a player physically collides with a station while moving, the client will need to detect the intersection and push the player back, preventing the player getting stuck in the station. In XKobo when a turret node is destroyed any children that the node had are also destroyed. The explosions travel up the tree, along the pipes, towards the core until they reach the core or another turret node, at which point the destruction stops. For this project I will alter this mechanic slightly so that players can only destroy nodes that are exposed, in that they have no child nodes. If the player attempts to shoot nodes that have children such as the core of an undamaged station, the collision detection will evaluate this and not cause the node to destroy.

4.7 Seamless level wrap

If a player crosses the boundaries of the level, the client will move player to the equivalent location on the opposite side of the level and players can see entities at the other side of the level from the boundary.

Not unlike classic arcade games such as PAC-MAN and Asteroids, XKobo has seamlessly looping levels. Although finite in dimension, the level wraps around which enables players, projectiles and other entities to move beyond the level boundaries and appear at the other side. For this project I will also include seamless level looping as a feature, and players, projectiles and stations will need to be designed in a way that allows their positions to wrap with the level. This level design allows confining the players to a smaller than usual area increasing the interactions between individual players and stations, but the seamless design of the level gives the illusion that the level is infinite and enables quicker maneuverability around the level. In addition to being able to move across the boundary, if the player is near to the boundary they should be able to see entities beyond the boundary that are on the other side of the level.

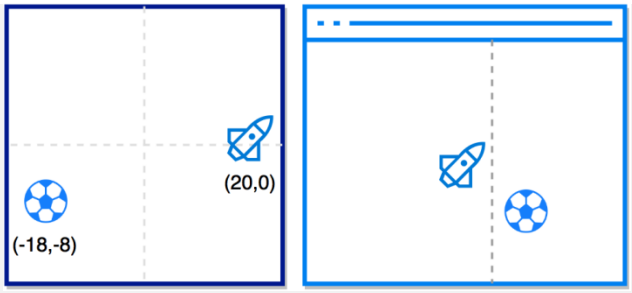

Figure 2 - Seamless level wrap. Left: actual positions of entities in the memory. Right: The player's perceived entity position in-game. The dotted line is used to show where the level boundary is and is invisible to the player.

4.8 Synchronization using communications protocols

The client and server will utilize communications protocols to send, receive and parse messages between themselves to achieve synchronization of the game state.

This is an understandably broad requirement and encompasses all of the functionality required for clients to play together in a synchronized network game. Both the client and the server will have the ability to create and send messages to each other. These messages vary in content and need to be labelled with their specific roles in order to be parsed correctly by the receiving client or server. Only a small proportion of the total data that clients store should actually be synchronized and it should only be data that it cannot calculate for itself. For example there is little reason in sending the textures of a player over the network to another client since every client has access to the graphical assets, and they do not differ during gameplay. In another example, projectiles travel in straight lines from their point of creation. Instead of synchronizing the position of the projectile frequently during its lifetime, one can simply synchronize where it was created, and its speed and direction since it will follow the exact same path on all clients. In other words, if clients are able to calculate aspects of the game by themselves like physics and motion in an identical fashion to how it would appear if it were synchronized, the former is much more desirable as it frees up bandwidth by pruning unnecessary messages.

4.9 Station synchronization

The system will synchronize the state of enemy stations across all connected clients so that when they are destroyed this event is relayed to all clients, and when a new player joins they receive the current state of all stations in the game.

Stations will to be synchronized across clients when any changes are made to their structure. For example, if a player destroys a node of a station, this needs to be relayed via a message to all other connected clients so that the specific node that was destroyed can be removed from the station data structures on the other clients also. All clients must have up to date knowledge of all the stations, not only so that players can cooperate in destroying the stations during gameplay, but also in the event that a new player joins the server and requires not the original, undamaged stations, but their current, possibly damaged state.

4.10 Hosting and handover

The server will designate a connected client to be the 'host', whose responsibilities include generating new stations to replace those destroyed and sending these to the server to be synchronized.

The client will have support for offline (non-network connected) play and therefore must not depend on the server for game logic such as station generation. If stations are generated client-side as opposed to by the server, then a single client among all connected to a server (ideally the first that connects) must be responsible for creating new stations and sending

them to the server. The server must then check that the client it is receiving the station from is the host, before incorporating the received station into its collection. In the event that the host disconnects from the server, no new stations will be created as the remaining clients are not hosts. In order for the game to continue, the server must allocate a remaining connected client to be the new host, which will then have the ability to create new stations. Otherwise the remaining connected players, despite still being able to move around and shoot, will have no new stations to destroy and the game is essentially in limbo until the server is restarted.

4.11 Performance and stability

Both the server and the client will maximize responsiveness at all times, without being overly taxing on the computer's resources. Both the client and server will avoid and handle exceptions as much as possible in order to reduce de-synchronization and prevent harm to the user's experience.

The user interface of the client, including the menus and in-game actions will have response time minimized as much as possible. For video games it is increasingly important that input such as key presses or mouse clicks are processed straight away as any delay in the user being able to see visual confirmation of feedback of the input can be frustrating and can make the user question whether their input was registered. Due to the nature of the game, quick response times of the client will be necessary to avoid incoming projectiles. Video games are extremely user-centric, and their inherent quality is often perceived as the quality of the user-experience regardless of any impressive technical mastery under the hood. A game with un-responsive controls or is therefore often considered of poor quality. If the client or server is programmed in such a fashion that requires a large portion of the available computational resources to run, not only would the client itself suffer response issues if the computer is not powerful enough to run the software, but also other applications running on the user's machine can suffer as a result too.

An important part of developing any software product is avoiding behavior that can cause the program to terminate unexpectedly due to an exception, and anticipating where such events may happen and compensating for them. For this project, an un-handled exception on the client side will significantly harm the usability and user-experience of the product since the player loses all progress gained during the game and will need to restart the client and reconnect to the server should they decide to continue playing. An un-handled exception on the server-side has the potential to be even more disastrous; since the server manages the synchronization of the game state and all clients depend on the server to propagate their messages to each other, a server crash would result in not only the loss of the session data that the server stores but also renders the clients unable to communicate with each other, effectively ending the multiplayer session permanently. As a result, the both the client and the server will need to be tested thoroughly to identify where these exceptions occur.

5 Technology

5.1 C#

Having previously developed games in Java using the Abstract Window Toolkit (AWT) graphics API, I sought to use the project as an opportunity to learn a new programming language without subjecting myself to learning a language with a different programming paradigm (i.e. non-object orientated) or choosing a language that is syntactically very different to Java, which I am very familiar with. I initially considered C and C++ (C being the language the original XKobo was written in), but despite being very popular languages for game programming, my time available to work on the implementation would likely have been dominated by having to learn new syntaxes and different programming paradigms.

5.2 Microsoft XNA / MonoGame

Microsoft XNA is a framework based on the .NET framework with the primary focus of providing a simple application structure and runtime for game developers. XNA does not provide much in the way of actual game logic and therefore it is important to make the distinction between a full-featured game engine, which can facilitate in writing game logic like Unity, and a much simpler and less-featured framework like XNA. Using XNA as opposed to engines like Unity has allowed me to have much more control and responsibility over the content of the game and allowed me to understand and implement common game logic myself without providing too much out of the box.

The MonoGame framework originally appeared in 2009 as an open-source version of XNA, but can be considered its successor as XNA was discontinued by Microsoft in 2013. MonoGame, unlike the original XNA, is multiplatform. I chose to migrate the project to MonoGame after a few weeks of development, first and foremost because MonoGame provides all of the same functionality of XNA (it essentially builds on the XNA code already present) whilst also being maintained and updated regularly. Despite the base XNA being quite mature and feature-complete, I was less inclined to continue using discontinued software and wanted to learn more contemporary development. Using MonoGame altered very little in the context of development and structure of the software, and is virtually XNA with a few added features.

5.3 Lidgren.Network

Lidgren.Network is an open-source networking library facilitates connecting clients and servers to transmit messages between them. The Lidgren library is present in both the client and the server as it is required for them to transmit messages between them. Lidgren is low-level, in it only supports the transmission of primitive types as opposed to more advanced higher-level libraries that support entire objects. This made it ideal since the focus of the project was on designing and implementing custom communications protocols and Lidgren was low-level enough to give me full control over the message content and delivery method (UDP/TDP).

5.4 Windows Presentation Foundation (WPF)

WPF is a modern framework used to create native Windows applications and is the successor to Windows Forms. It allows developers to construct interfaces using pre-defined elements such as buttons, text areas and scroll boxes in XAML. This layout is parsed and displayed by a C# program containing the separated business logic that defines behavior when elements are interacted with, or the information the interface displays. Originally the server application for the project was a simple C# command line program. Late in the development phase I chose to migrate the code to a WPF application. WPF allowed me to display the game state in a much clearer manner, with separate list panes displaying players and game events.

5.5 Visual Studio 2015

Both the client and server software was developed in Visual Studio 2015. The MonoGame software development kit (SDK) integrates with Visual Studio and allows access to the functionality that the framework provides. Visual Studio also contains a debugger like many IDEs that assisted in identifying the nature of exceptions. Integrated software metrics tools allowed me to calculate interesting metrics such as cyclomatic complexity and maintainability indexes, which gave me feedback on the quality of my programming.

6 Network Architecture

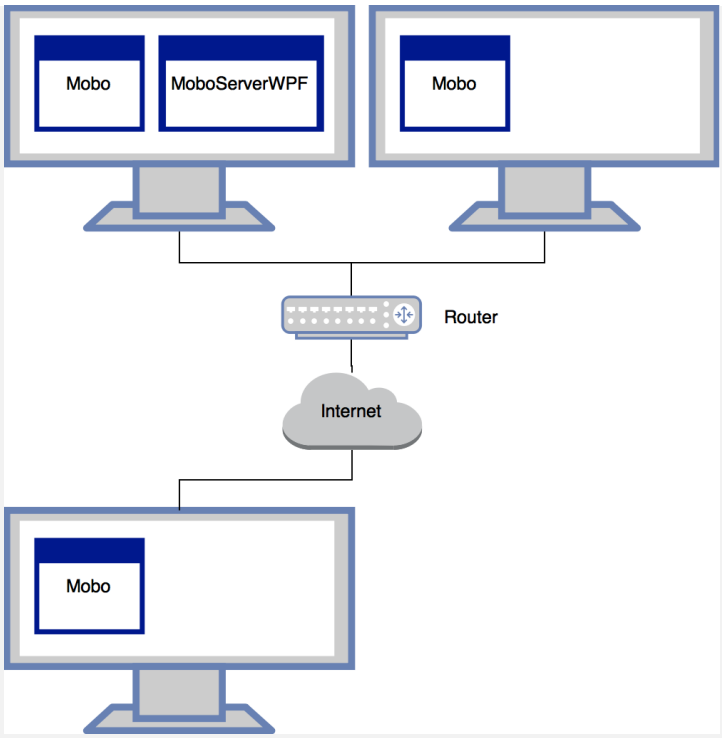

Figure 3 - A computer network diagram of three Mobo clients connected to a MoboServerWPF server instance.

The project was implemented using the client-server architectural model. Mobo and MoboServerWPF are names of the separate client and server applications respectively. Lidgren.Network allowed me to relatively easily implement the client-server model by instantiating a Lidgren.Network `NetServer` object on the server, and a `NetClient` object on the client, and giving them the same configuration.

The diagram above shows three methods Mobo clients can connect to the server. Starting both the client and server and configuring the client to search for the server with an IP address of 127.0.0.1 (known as localhost) will point the client to open a connection to the server instance running on the same computer. This was extremely useful for testing since the server and client could be ran alongside each other on the same computer without requiring two separate computers. In the top-right of the diagram, a Mobo client connects to the server instance from a separate computer, achieved by using either the local IP address of the computer running the server (LAN) or the IP address visible to the rest of the internet (WAN). Finally, any client running on a computer across the internet is able to connect to the server instance running on the top-left computer by configuring the client with the host machine's public IP address. Any permutation of these different connection types are supported by Lidgren.Network, assuming that the port the server is running on has been correctly entered in the client, and that no firewalls on the server-side prevent outgoing or incoming data on that bound port.
The server could have potentially been a component of the same application as the client. Indeed, many games with network functionality employ this design; the client and the 'listen'

server execute in the same process and the user chooses whether to host or join a game. If the user hosts a game, the program assumes the role of both a client and a server, allowing external networked users to connect to the program to play in the same synchronized game as the host. If the user opts to join an existing server, the server component is not activated and the client proceeds to connect to an external server. Listen servers are popular in the games industry as the software can be packaged as one complete deliverable and does not require the user to run more than one application. Most modern, high budget titles often have the ability to connect to both dedicated and listen servers.

Despite this, I decided it would be more favorable to create separate applications for the client and server. Firstly, keeping the system modular would enable development and debugging of the client and server independently and increase the simplicity of each application. In addition, keeping them as separate applications with no shared code would prevent any chance during the development of the project that I could accidentally couple the client and server components too strongly and end up straying from the client-server model. Secondly, as a personal addition to the original project proposal provided, I decided that the server would resemble a distributed dedicated server. Dedicated servers run independent of their counterpart clients and can continue to run even after all players have disconnected.

The benefit of such a server is being able to support the ‘drop-in-drop-out’ concept of multiplayer gameplay; new players are able to join at any time and receive an up-to-date and current version of the game state, and players are able to leave freely without disrupting or destroying the game state for other players that may still be connected. With the non-dedicated server design that some games employ, if the player hosting the game closes the game application, encounters a fatal exception or otherwise the network connection between his/her computer and the other clients that are connected fails, none of the previously connected clients are able to continue playing because the server was responsible for synchronizing the clients and managing the game state.

For the specific architecture of the technology behind the client and server, XNA applications and Windows Presentation Foundation applications respectively, see Appendix 14.1 and 14.2.

7 Implementation of the client

7.1 Main type

The interface that XNA provides for structuring the main class of the client, `Game1.cs`, ensures the implementation of five important methods integral to the runtime of an XNA application. Appendix 14.3 shows a simplified version of the main type of an XNA application.

The constructor is used to configure aspects of the application before any game logic has begun to execute. It is here that I set the root directory for the content (XNA's name for assets such as graphics and audio) to be loaded later, initialize the class I wrote to manage the user settings of the application, `SettingsManager.cs`, and load them from the settings XML file and initialize the `GraphicsDeviceManager`, a component of the XNA Core Framework layer. The primary role of the `GraphicsDeviceManager` is to enable configuration of window-specific options such as title and resolution read from the `SettingsManager`. Here I also set the application to hide the Windows system cursors that normally overlay XNA applications since I implemented my own set of cursors that change graphic contextually.

The `Initialize` method contains operations related to the initialization of the graphical user interface of the client. Since I have written the client to enable changing the resolution of the game if the user so desires, here I calculate the relative center of the screen and store it in the custom static class `ScreenManager` so that GUI elements and logic throughout the application can use the center of the screen to position and center elements. Initially, all GUI elements were positioned using absolute coordinates for the default window size and as a result, when I added the ability to change the window size, GUI elements, especially those in the menu and the settings screen appeared misplaced, signifying the need to consider the screen size when positioning the GUI. Since the `Initialize` is only run once in the lifetime of the application, the client needs to be restarted after changing the resolution to see the effect.

The `LoadContent` method, seen in Appendix 14.4, is responsible for loading in the graphics and audio assets of the game and storing them in `Texture2D` objects. A `SpriteBatch` is an XNA framework provided type that allows the developer to essentially append draw calls for graphics to a queue so that they can be drawn all at once, with the same settings. `SpriteBatch` types have a number of configuration options, such as enabling tiling graphics for backgrounds, stretching graphics over an area, anti-aliasing and the Z-axis order in which the graphics are drawn. `ContentStore` is a custom type that handles the loading into memory of all graphical, audio and font assets. It stores the assets in static `Texture2D` objects that can be accessed by any class where required.

It is common for XNA games to only require one `SpriteBatch`, especially if the viewport, the user's visible area of the game environment, does not change. It became necessary for me to utilize two `SpriteBatch` types once I began programming the in-game environment due to the necessity to distinguish objects in the game such as players, projectiles and stations which

constantly change position depending on the viewport, and static GUI elements that never change position such as the health bar, player list and message log. Using two `SpriteBatch` types allows me to manipulate and transform the viewport using `cameraSpriteBatch`, which I programmed to follow the user's player around the level by setting the viewport to the player's current position, and drawing static UI elements with `staticSpriteBatch`.

`GameState` is an enumerated type I created to represent the current state of the game. For instance, there are different states for the main menu, settings menu, help screen, offline play and online play. The idea behind game states is to notify the `Update` method of the main type what logic it should be executing based on the current context of the game, and notifying the `Draw` method what the user should be seeing depending where they are in the application. During the runtime of an XNA application, the program repeatedly executes the `Update` and the `Run` methods sequentially (together known as a tick), so structuring my application in this way has allowed me to create encapsulated classes that define exactly what logic should be executed, and what should be shown to the user, only when a particular game state is active.

7.2 Menus

Figure 4 - The client main menu, shown at the startup of the application.

The client main menu provides all available functionality to the user. 'New Game' sets the game state to 'Offline' and starts a new single player session (non-networked). 'Join Server' sets the game state to 'Online', invokes the Network class to instantiate a Lidgren.Network `NetClient` object and attempts to open a connection to a server using the IP address and port specified in MoboSettings.xml. 'Settings' direct the user to the settings menu that allows them to configure options such as player name and window resolution, and Help directs the user to a simple screen showing a graphical representation and hints of how to play the game properly.

Figure 5 - The settings menu, featuring a few unused entries.

Figure 6 - The help screen, designed with various screenshots from the game.

As mentioned in section 7.1, the `SettingsManager` static class is instantiated at the start of the runtime of the application in the constructor of the main type. `SettingsManager` makes use of the native .NET XML API to read in the user-configurable settings of the game from the settings file MoboSettings.xml, found in same directory of as the client executable. Currently, clicking most of the options in settings that require text input like name open a new notepad instance where the user can edit the XML directly. Such a method is undesirable in my opinion, but since XNA does not provide any components that allow for text input, I would have had to program a text editor myself and I felt expending effort on such a task would distract from the focus of the project and my development time could be better spent.

The buttons (signified by a cyan outline) and the fields that show values (grey outline) are of my own design and implementation, and necessary since XNA does not provide any GUI functionality. Each encapsulated menu class such as `MainMenu` or `Settings` contains a list of `Button` and `Field` types. In the `Initialize` method of the main type, all the menu classes are initialized (this can be seen in Appendix 14.4). During this initialization, all the `Button` and `Field` elements are instantiated and their positions relative to the screen center. In the case of buttons, the text they display is defined and with fields the data that the `Field` binds. If the particular game state is active, the menu class calls `Update` on all `Button` and `Field` classes to check if the user has clicked any and update any new value of the variable respectively.

Considerable effort was put into the design and aesthetic of the menu GUI. The textures that the Button and Field types use, the background and the cursors were sourced from free and open source websites, like many assets used in the project. The cursor, when hovered over a button also changes contextually, and this is achieved by creating an enumerated type `CursorState`, that works in a very similar fashion to that explained of `GameState`.

7.3 In-game environment

Figure 7 - A in-game screenshot of the Mobo client (online mode).

The figure above shows what the user sees during the online gameplay, but is almost identical to that of the offline game with the exception of the player list. The health bar at the top left, the mini-map, center top, the message log, bottom left are drawn using the static `SpriteBatch` since they should remain in the same positions. All other elements seen above, like players and background are drawn using the camera `SpriteBatch`, which moves the viewport based on the position of the local player. If the player moves up, the background will scroll with the player as they move and stations will appear on the screen once the player is close enough to see them, as is true with other players. The viewport is offset slightly in the position of the cursor and this was implemented to reveal more of the game environment in the direction that the player is actively aiming in.

The dimensions of the in-game environment are 3600 by 3600 pixels as shown by the area that background above covers in Appendix 14.5. Background is a custom class I wrote to provide a background for every game state, given a `Texture2D` is provided as a parameter for all menu classes and is instantiated with each menu class in the main class of Mobo. The background is drawn at position (-1800, -1800) in this instance because the player spawns at the center, position (0, 0). At the center, the boundaries of the level are therefore 1800 pixels in each cardinal direction.

In Appendix 14.6, one can see that the principle entities of the game, Player types and Station types are stored in `ConcurrentDictionary` types within the in-game game state classes (both `Offline.cs` and `Online.cs`), the latter being encapsulated within a `StationSpawner` type. `StationSpawner` was a late addition to Mobo but vital in managing the position and 're-spawning' of stations if they are destroyed by players. `StationSpawner` simply checks the number of stations present in the station dictionary and creates new stations until there are 3

total in the collection. Storing the players and stations in easily accessible, iterable types was required because many classes iterate over the player and station list to achieve their functionality, oftentimes modifying the collection during the foreach loop. For this reason, the data structures had to support concurrent access and write as many classes read these types while they update as players move around and interact with stations, such as `Minimap`.

7.4 Mini-map

The code in Appendix 14.7 shows the code that draws players in the correct positions on the mini-map. The variables `height` and `width` define the size of the mini-map itself, and is also the size of the texture overlay shown in Figure 8 that gives the mini-map the appearance of radar. The range of the mini-map is the maximum range in any cardinal direction from the position of the player that the radar can detect and display players.

During gameplay during every tick, or frame drawn, the `Draw` method of the mini-map invokes two methods defined in its class, `DrawPlayers` and `DrawStations` for drawing the players and the stations respectively. Both methods have near-identical operation, with `DrawPlayers` iterating through a list of the players in the current game and `DrawStations` iterating through the flattened list of nodes for each `Station`, so for the purpose of avoiding repetition only `DrawPlayers` will be further explained.

For each player, the difference in x any y coordinate from positions of the iterated player to the local player is calculated. Note that since the local player is a member of the dictionary of players itself, at least one run of the foreach loop results in the local player being compared to itself. This could have rather easily been avoided by a condition check to make sure the iterated player is not the same as the local player, but I ultimately decided to leave this in as it results in a dot being placed at the center of the mini-map representing the local player, simulating the player being able to see his/herself in the center of their own mini-map.

The if condition checks if the relative difference in x and y coordinates lies between the maximum ranges in either Cartesian direction, and if so, the player is drawn on the mini-map. If this range check were not present, players and stations would be drawn on the mini-map regardless of the relative distance away, resulting in dots being placed well beyond the visible radar graphic. `gridx` and `gridy` store these the coordinates, `dx` and `dy`, converted from their Cartesian coordinate system to the coordinate system used by computers, in which the origin is in the top left. They are also divided by twice the range to transform the coordinates to coordinates values between 0 and 1, and then multiplied by 128 and cast as an integer to achieve a pair of coordinates that can be placed on the 128 by 128 pixel mini-map. Using these coordinates, the players and the stations are then drawn in the correct relative positions within the boundary of the mini-map. Subtracting 1 from the final positions of the dots centers them correctly about their coordinates (dots have a width and height of 2).

7.5 Players, movement and projectiles

The `Player` type encapsulates all of the functionality of player entities, the spaceships. Every instance of the client has a local player, that is, the one that is controllable by the user. A large proportion of the functionality in the `Update` method of `Player` is exclusive to the local player. By separating the functionality of local players and other players, we can ensure that the player can only control and manipulate the behavior of the local player bound to their client. The `Update` contains many calculations that take place per tick, which includes checking for input from the mouse and keyboard, movement and orientation, firing projectiles, collision detection and updating the projectiles that belong to the player.

The user can move the player character by pressing the directional keys or using WASD. The first statement in the `Update` calls the `HandleInput` method of the same class which obtains a current version of the keyboard state so it can check for inputs. Each player has a `Vector2` object that represents their current velocity. If a player presses the right directional key, `HandleInput` increases the x-coordinate of their velocity by 0.2 per tick as long as it is pressed, and if the left directional key is pressed the x-coordinate of their velocity is decreased by 0.2 and so on and so forth. Each tick the velocity is added to the current position of the player, and thus the player moves if their velocity is greater than (0,0). The value of 0.2 was chosen as ideal after a little trial and error, larger values made the player too fast and uncontrollable and smaller values made the player frustratingly slow. Finally, the velocity is multiplied by 0.96 each tick so that the velocity slowly decreases to near 0 if the player isn't pressing any keys to simulate friction. Obviously there is no friction in space, but if the player is constantly moving despite not pressing any keys it can be difficult to aim and made moving around feel slippery and uncontrollable.

Unlike XKobo in which the player was confined to 4 directions of movement, players in Mobo can move in a full 360 degrees of movement. It is therefore necessary to angle the player sprite in the direction of movement if the player is moving, or the direction of the crosshair cursor if the player has released the keys and is aiming with the mouse. Appendix 14.8 shows the code responsible for rotating the player correctly, based on the context of gameplay. The two class variables `playerRotation` and `mouseRotation` store the angle based on the player's velocity and the angle from the player to the crosshair cursor. This not only produces a nice visual effect as the player sprite rotates as it changes direction, but is also important for calculating the velocity and direction of a projectile when the player shoots.

Appendix 14.9 shows the implementation of these methods. The calculate direction method uses the velocity of the player and the inverse tangent function of the `Math` library to calculate the resultant direction the player is travelling in. For example, if the player has a velocity of (2, -2) we know they are moving a positive 2 pixels in the x-axis (east) and a negative 2 pixels in the y-axis (north). The resultant direction produced by `calculateDirection` would be a value in radians that represents a north-east direction. Adding Pi/2 is necessary to correct the direction, without out it the player seems to face 90 degrees clockwise to the real angle of movement. The second method, `angleToMouse` is used to calculate the direction of the cursor,

relative from the center of the screen and is used when not moving to enable the user to aim. First, the positional difference is cursor to the center of the screen, and (16,16) is added to shift the center of the cursor to the middle of the crosshair graphic. The resultant angle points in the exact direction of the cursor relative the center of the screen.

When the left mouse button is pressed (handled by the same method as movement) the player fires a projectile in the direction that `angleToMouse` calculates each tick. To calculate the necessary velocity of the projectile, the opposite of the above calculations takes place. The angle of the player is converted into a vector by using trigonometric functions Cosine for the x component of the velocity and Sin for the y component, shown in Appendix 14.10.

Projectiles move in an identical fashion to players, by adding current velocity to the position, but their speed is constant and they do not slow down. Sin and Cosine output values that are between -1 and 1, which proved far too slow for a projectile, so this is multiplied by a constant `bulletSpeed` with a value of 10. Each Player object representing an individual player in the game has a class variable of type `ConcurrentDictionary` that stores all the projectiles that belong to (i.e. were fired by) that player. I designed them in this way as opposed to a single object that stores all projectiles stored by all players so that the owner of the projectile can be easily determined. This was useful for rewarding players with score as they destroy stations, and is explained in section 9.4.

Finally, the `Player` class is responsible for local player specific collision detection. All entities in the game that can be damaged, or cause damage have instance-specific bounds, that update every tick to follow the entities. Bounds are essentially `Rectangle` objects that define the collision boundaries of objects. In Mobo, the bounds of most entities are slightly smaller than the dimensions of the sprite. This is to allow projectiles to ‘graze’ players and not hurt them. During testing, when the bounds were set at the exact dimensions of the sprites, sometimes one would take full damage from a projectile that should have barely missed by a pixel or two, so reducing the bounds dimensions prevented this from occurring. The .NET framework provides the method `Rectangle.Intersects` which I used to check for collisions between bounds. Each client checks only for the collisions made to its local player by iterating through all other player projectiles, and checking if the bounds overlap. The settings menu includes an option to display all bounds, which I added so I could better determine the size of each bound relative to its sprite.

7.6 Station structure

Figure 8 - A station as they appear in Mobo.

`Station` is a large complex class that represents the enemies in the game. A station's structure is modelled as a tree; each node in the tree is represented as `Node` types which have a parent (except in the case of the core) and can have multiple children. Since .NET does not provide any tree-related classes (unlike Java that have `TreeNode`) it was necessary to implement the nodes and their functions myself. Each `Node` type, that represents a node in the tree of a station, has a single `StationNode` object that handles the behavior, appearance and type of the node. Figure 9 shows a station made up of multiple `StationNode` types.

The square node near the bottom of Figure 9 with the 'smiley face' represents the core of the station. There are a series of meandering 'pipe' `StationNode` types before the tree splits into two. Pipes only serve to connect turret and core nodes, and will destroy themselves if they do not have any children (i.e. they do not connect to anything). The circular `StationNodes` are the turrets, and if they are exposed (i.e. they do not have any children) they will turn red and begin firing at all players within their range.

Appendix 14.11 shows a simplified structure of a typical station. By design, only the exposed (red) nodes can be destroyed by player projectiles, shooting a node that is protected by children will have no effect. This encourages players to target the leaf nodes of the tree, destroying the station gradually before destroying the root node, the core and thus destroying the station.

I decided to make `Node` and `StationNode` separate classes, despite them having a 1 to 1 relationship, in order to distinguish the code used to model the tree structure, and the entity `StationNode`, that has a has a sprite, and other class variables similar to Player like health, bounds and projectiles. This ultimately was a good decision I believe as it abstracts code that

contains game logic and calculations, like that found in the `Update` method of the `StationNode` class, from the code that `Node` contains which defines common tree methods such as `Add` (as child to a node), `Remove` (a child from a node's children) and return parent. StationNodes also have very basic 'AI'. Using trigonometric methods seen in the `Player` class, StationNodes shoot players that are within a range of 540 pixels. This is achieved by calculating the distance between the each StationNode and all Players, and if this distance is less than 540 the StationNode fires a projectile using a similar trigonometric method as is used when a Player shoots. A cooldown timer, identical to that of Players prevents the StationNodes from firing too quickly.

All `Station` objects contain a flattened list (yet another `ConcurrentDictionary` type) which contains references to all the `Node` objects in their station tree. The advantage of having a flattened list is being able to remove specific elements with ease without having to parse through the entire tree that could be large, and to update and draw the StationNodes I only needed to loop through the list using a foreach statement and call `Update` and `Draw` on each element.

7.7 Station generation

Station structure is randomly generated using an algorithm that first randomly generates the tree structure using `Node` types and then attempts to build the generated tree in 2D space starting from the root. There are four important parameters in the constructor of `Station`, shown in Appendix 14.12, that influence the structure and size of the station:

`split_depth` refers to the interval at which nodes will split (have more than one child). For example, if the maximum depth of the tree is 15 and the split interval is 6, the tree will branch twice, first at depth 6 and all branches from that point on will branch again at depth 12. `max_depth` refers to the maximum depth of the tree and the algorithm will continue adding children until it is reached. `difficulty` is a set value between 1 and 10 that influences the number of children at each split. When the difficulty is 1, there is a high chance that at a `Node` will only have one child, and a rare chance that it will have more. Conversely, when the difficulty is 10, there is a high chance that a station will have 3 children (the maximum for a non-core node), and a rarer chance that it will only have one or two. Difficulty is found in the settings, but due to time constraints and focus on more important aspects of the project, I was unable to manifest it in the station generation. As a result, by default, all stations have a difficulty value of 5. Finally, `initial_branches` is the maximum number of branches that extend at the start from the core. Originally designed to behave similarly to difficulty, during testing I found initial branches to be the single biggest influence on the station size and complexity, so I ended up enforcing the value as 2 for all stations.

Generating a station begins by creating the root node, `station_tree`. The parameters that `Node` takes are the depth, 0 since it is the root node, and the parent node, which is null since the node has no parent. The data variable is the single `StationNode` type that is associated to the node. It takes its coupled node reference as a parameter so it has access to the tree functions of its node, and also the entire station that it belongs to as a reference so it is able to remove itself should it get destroyed by a player. Each node also has a `Vector2` position. This isn't its position of the Node in the game environment; instead this is the position of the Node relative to the Core which is used to build the station in 2D space. This position is finally added to a `HashSet` of `Vector2` objects. This is to prevent nodes stacking on top of each other. By keeping track of previous locations where nodes were built and checking the `HashSet` before building to see if a node at a certain position exists, nodes can no longer stack. Before this was implemented Stations were quite small in size, it was only when I realized that certain positions were occupied by more than one did I realize that it was necessary to implement this check.

The station tree is then built recursively from the root using method `GenerateNodeChildren` shown in Appendix 14.13. `GenerateNumNodes` simply returns a number that represents the number of children the current node should have. Unless the depth is at the split interval, it will always return 1. When the node depth % split depth = 0, `GenerateNumNodes` will generate a number based on the difficulty. If the depth of the tree is at the max depth, the algorithm will terminate and return the tree. If not, the method then recursively calls itself for as many

nodes are specified by `num_nodes` in order to continue recursively building the tree on all branches, adding one to the depth each time a successive `GenerateNodeChildren` is called.

Now that a station tree has been created from `Node` objects, the station has to be built in 2D space to resemble that in Figure 11 below. Appendix 14.14 shows the `BuildStation` method that achieves this. `BuildStation` is a recursive method that takes the entire station tree as input excluding the root and designates positions for each node in a 'grid'. The `HashSet poss` contains a list of grid positions that the algorithm will attempt to build in, essentially, one unit vector in each cardinal direction from the current node's parent. To increase the variability of the build algorithm, the list is shuffled. The algorithm then takes the parent node's position, for instance the core at (0,0) and adds the first possible direction in the shuffled `HashSet`, for example (1,0). The resultant vector, (1,0) is then checked against the master `HashSet` that contains the positions of all nodes in the station. In this example since the only node built so far is the core, the only position in the master `HashSet` is (0,0) so the node is free to build at (1,0). If it were the case that a `Node` already exists in the calculated position, the position vector that a build attempt failed with is removed and the algorithm attempts to build again with a new position. If the grid space is free, the new position becomes the position for the node and a `StationNode` is initialized. The position of the node is finally added to the master `HashSet` to prevent other nodes being built on the same 'square', and the algorithm continues for the node's children.

For every node that has been built in a valid position, a new `StationNode` is created that represents the appearance and behavior of the node in the tree. The root of the tree is always a core `StationNode`, every `StationNode` with a depth of the split interval + 1, and the leaf nodes of the station will be turrets, and the remaining StationNodes are 'pipes'. The correctly angled pipe texture is assigned to each StationNode by examining its parents and children, and the position they are relative to the current StationNode.

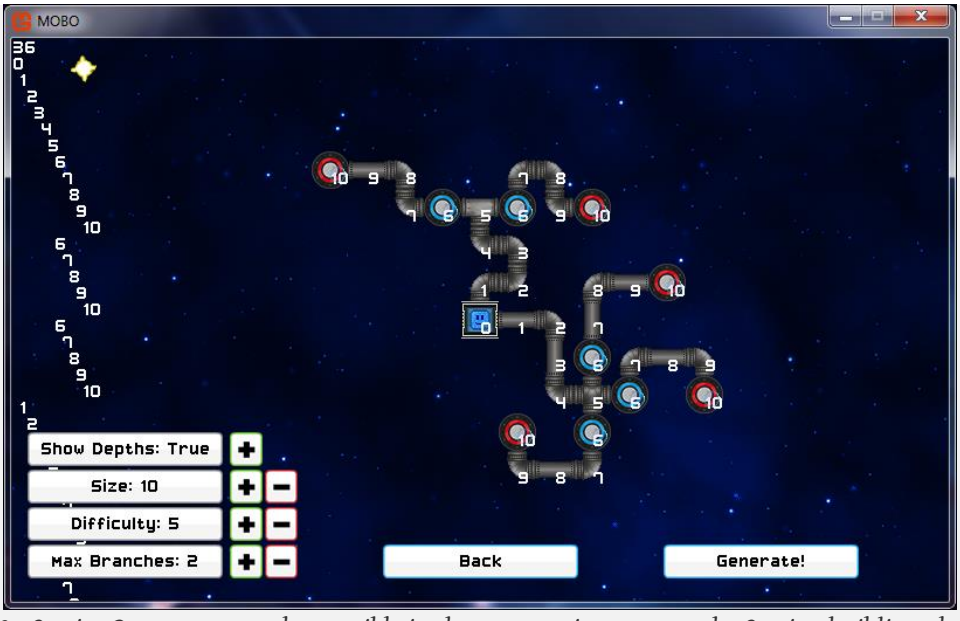

Figure 9 - StationGenerator, a tool accessible in the menu written to test the Station building algorithm.

8 Implementation of the server

8.1 Graphical user interface

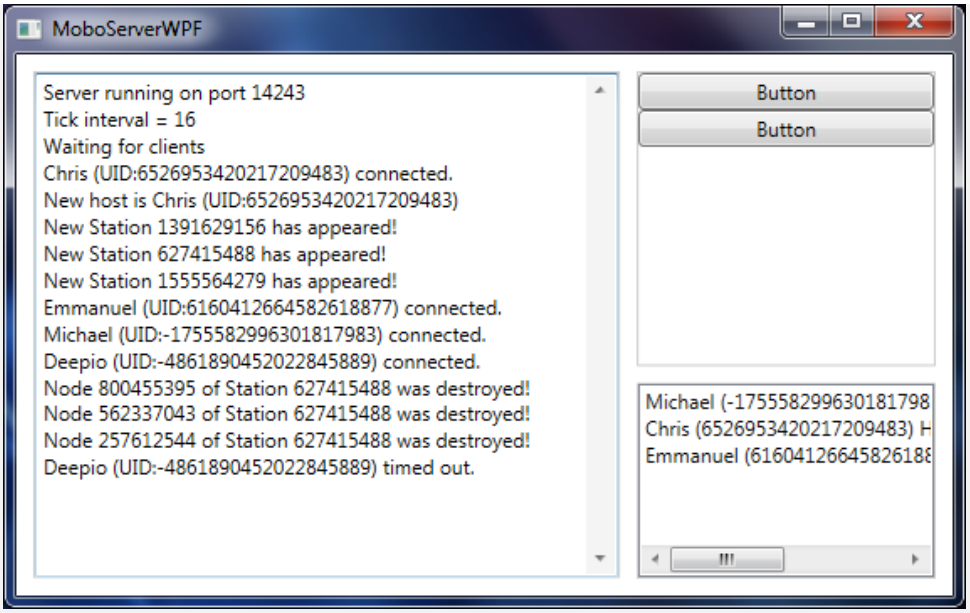

Figure 10 - A screenshot of the server application, MoboServerWPF.

In comparison to the 'thick' client Mobo, which boasts 35 classes, the server requires just 4 excluding the main type. The server calculates very little game logic itself; its primary role is to act as a receiver for incoming messages, parse them based on the message type, modify its own data model accordingly if required, and if the protocol requires it, distribute the changes to the connected clients. The GUI is defined from the layout XAML for the WPF application, MainWindow.xaml. Having previously designed GUIs in Windows Forms, the benefits of WPF over Forms were strikingly apparent. Since all the layout and elements are defined in the XAML, the remaining code is business logic. In Windows Forms, much like `JPanel` in Java, the code can consist of large sections of presentation logic and business logic intertwined, making WPF a much more attractive solution.

The `TextBox` element on the left contains an almost complete log of the notable events that have transpired during the lifetime of the server and automatically scrolls as new events occur. The `ListBox` in the bottom right contains a list of the current players in the server, and scrolling to the right reveals, their unique client identifier (UID) used to distinguish between players, their health, score and whether they are the host. The buttons in the right pane do not have any function and I added them in case I would need some way to interact with the server directly, for example to kick (remove) players from the server or otherwise interact with the game state. Ultimately this fell out of the scope of the project and was not implemented.

The main type of the MoboServerWPF, `Program`, has a constant called the tick interval (tickrate in the code) that determines the rate at which the server will read messages, update its data

model to include new data, and distribute new messages to clients. A tick interval of 16 (60hz) means that every 16ms the server will parse the collection of messages it has received and behave as defined in the message type protocols. If the tick interval is too large (meaning a greater time passes between each refresh) the clients would receive important messages such as the positions of players too infrequently, meaning the individual clients have a less accurate view of the current game state. Since the client runs at 60 frames per second, there is little reason to decrease the tick interval less than 16ms since the clients do not update faster than this and any additional messages between each tick would not be parsed. Every 16ms, implemented using a Timer object, the server updates a static `Network` class that contains the network protocols, and the Player class that contains a simplified representation of the Player class of the client, to store the players of the server.

8.2 Network class

The client and server share a similarly structured `Network` class, containing the protocols responsible for reading a stream of incoming messages and parsing them. The `Update` method of the `Network` class consists of a single switch statement that reads the byte header of the message (determines what type of message it is) and offloads the message to different methods accordingly. Each message type has its own method (the protocol), and in both the client and server, the `Network` classes have a single method for every message type. Appendix 14.15 shows the while that loop ensures the class deals with every message it receives between each tick. Reading messages using the Lidgren.Network library works similar to reading streams of data, in that primitive types are read sequentially from the byte stream by using the correct read method for the type.

8.3 Player class

The `Player` class contains a publicly accessible static list of `Player` objects that represent the players connected to the server. The `Player` class in MoboServerWPF and the `Player` class in Mobo are significantly different. As mentioned in requirement 4.8, ‘Synchronization using communications protocols’, only the data important for synchronization of players needs to be stored in the server. Therefore, the class stores only the client unique identifier for the player, positional related information (two ints representing their x and y coordinate positions and a float in radians for current direction), their health and score.

The `Update` method for `Player` has two responsibilities. Firstly, every server tick a ‘timeout’ counter is incremented for each client connected to the server. The timeout counter is reset to zero if the server receives a message from the server, showing that the client is still connected. If the counter reaches 180 for a client, i.e. it has been 3 seconds since the server last heard from a client, it is considered disconnected and the server closes the connection and distributes a message that causes the player to be removed from all clients. Secondly, vital for movement synchronization, the server sends the positional information for all players that are connected to all clients.

9 Synchronization

9.1 Overview

There are two common models for game server synchronization following the client-server architecture, authoritative and non-authoritative. Authoritative servers have total control and domination over the game state, and contain the vast majority of the game logic. Clients seek 'permission' in order to change the game state, for even simple changes like player position, and if the server rejects the change for any reason (a large increase in score may indicate cheating) the change is reverted on the client. MoboServerWPF follows the non-authoritative model, primarily because the client is 'thick' and contains the majority of the game logic. The server therefore acts as a message distributor and a game state tracker for new and current players.

This section details the protocols written to maintain the state of synchronization of all the elements of the game that need to be synchronized. Since the method for constructing and sending messages in the code is identical for all message types (an example seen in Appendix 14.16), the specific code will not be detailed for every message type. Instead, the role of each message type, the primitive type contents, and how they are interpreted by each `Network` class of the client and server will be discussed. The table below shows a summary of all the message types in the game, the role of the protocols, and whether the client and/or server send them.

Message Type	Byte value of header	Role	Sent from server to client? Yes/No	Sent from client to server? Yes/No
CONNECT	100	Connecting a new client to the server	Yes	Yes
MOVE	101	Updating the position and direction of a player	Yes	Yes
DISCONNECT	102	Disconnecting a client from the server	Yes	Yes
CREATE_PROJECTILE	150	Creating a new projectile at a positon	Yes	Yes
REMOVE_PROJECTILE	151	Removing a projectile from the game state	Yes	Yes
CREATE_STATION_PROJECTILE	152	Creating a new projectile belonging to a station	Yes	Yes
CREATE_STATION	160	Synchronizing a new station created by the host	Yes	Yes
REMOVE_STATION	161	Removing a station, for when they are fully destroyed	Yes	Yes
REMOVE_NODE	162	Removing a single node of a station when destroyed	Yes	Yes
HEALTH	200	Updating the health of a player	Yes	Yes
KILL_PLAYER	201	Killing and re-spawning a player if their health reaches zero	Yes	No
SCORE	202	Updating the score of a player	Yes	Yes
NEW_HOST	210	Delegating a client to be the host	Yes	No

Table 1 – All message types used in the project. The byte value for each message is arbitrary; it is only used to identify the type of message in the Network classes.

9.2 Connections and disconnections

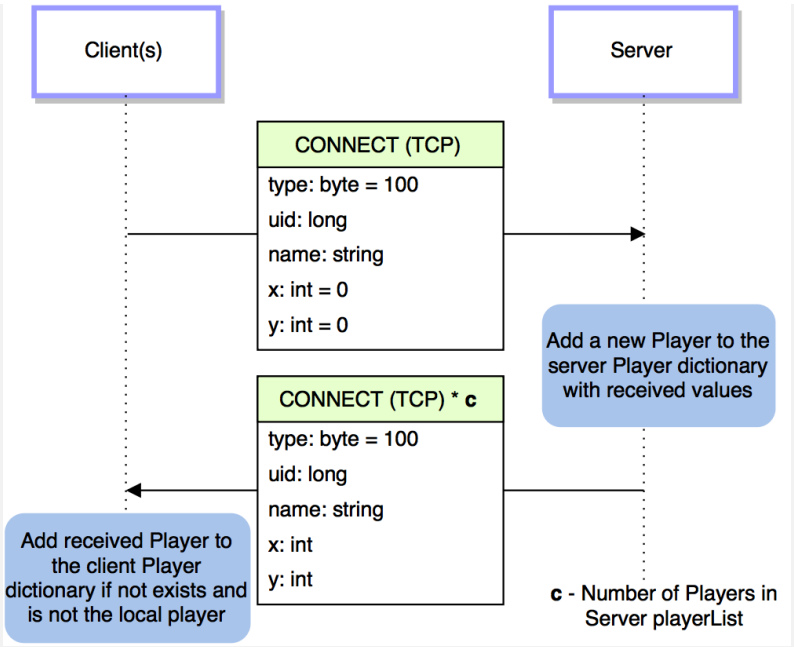

Figure 11 - Sequence diagram of a client connecting to a server.

When a new client attempts to connect to a running instance of MoboServerWPF, the connection is first established using a Lidgren.Network `NetClient` object with the correct IP and port configuration. Immediately after, the client constructs a CONNECT message that contains the unique identifier of the client (provided by `NetClient.UniqueIndentifier`), the name of the player and x and y coordinates of the starting position of the player. The server creates a new `Player` object for the player, updates its player collection and subsequently sends details of all players to all clients via a returning CONNECT message, for two reasons. First, previously connected clients need to add the new player their collections, and secondly, the newly connected player requires the details of all the current players connected to the server. By only adding the player to the client collection if it does not already exist (determined by the unique identifier), both of these requirements are satisfied.

Disconnections are achieved in a similar fashion. If a client is terminated or otherwise loses connection to a server, the client will time out on the server side as explained in section 8.3. When this occurs, the server creates a new DISCONNECT message containing the UID of the client that has timed out and sends it to all clients. The clients receive the message and remove the player with the specified UID from their collections, notifying them that the player has left the server.

Not that the delivery method of CONNECT, as well as disconnect is TCP. In fact, all messages except MOVE are delivered via TCP, as they are important to the synchronization of the game state, and failure to transmit and receive any of these messages can lead to inconsistencies of the game state between clients. If MOVE messages fail to reach server or client, is it not such an inconvenience since they are transmitted by server and client every 16ms.

9.3 Player positions

Player positions and directions are synchronized by MOVE messages. During every tick, the `Player` class of the client will create and send a MOVE message to the server containing the UID, position (in separate integers like the CONNECT message), and the current rotation of the local player. Synchronizing the rotation of the player was important as otherwise, all non-local players appear to be facing north no matter the direction they are moving or firing. The server updates the player position and rotation in its collection with the specified UID and then sends a MOVE message to all clients identical to that it received from the individual client. The connected clients then check first if the MOVE message does not relate to their current player, and if not, update the player's position in their collections. Without this check, the client would continually update the local player with the position defined in their own move message, rendering them unable to move.

The code excerpt in Appendix 14.16 is the implementation of the MOVE message from server to client, showcasing the construction and transmission of messages using Lidgren.Network.

9.4 Projectiles

When a player fires a projectile, the client sends a SHOOT message to the server containing the UID of the local player, and positon. Since the server does not store information about projectiles, the server simply propagates the messages to all clients. When the clients parse the SHOOT message, a new Projectile object in instantiated and added to the owner's projectile collection (determined by the UID) and using the trigonometric methods defined in section 7.5, the projectile is given a velocity and appears in game. The message is ignored if it relates to the local player, preventing a duplicate projectile being added to the player's projectile collection. REMOVE_PROJECTILE ensures the removal of projectiles if they collide with StationNodes or other players. Stations have their own projectile messages as the projectiles belong to the StationNodes that fired them, but the operation is essentially the same.

9.5 Health and score

HEALTH and SCORE are similarly functioning messages that maintain synchronization of the health and score of players respectively. Both `Player` types and `StationNode` types have bounds used in collision detection. When a projectile, whether it be from another player or a station, intersects the bounds of a player a HEALTH message is sent to the server with the relative change that needs to be made to health of the player (most often -10 to simulate damage). The server then adds this value to the health of the respective player in its collection, and sends the absolute value to all clients. The health of the player is subsequently changed on all clients. SCORE works in a similar fashion, except they are sent by `StationNode` types when player projectiles collide with them. In order to allocate the score to the correct player, the player UID is sent along with the relative increase in score to the server, which then distributes the UID and absolute score to all the clients to award to the player for destroying a node.

9.6 Stations

Stations are inherently complex data structures. It is necessary for the server to keep track of all the stations and their structure and relay them to clients when changes occur or they otherwise require an update. In this way, possible de-synchronization is avoided as clients are not required to keep track of all the changes made to the station structure them, and the server ensures that all the clients maintain the same knowledge of the stations that exist and their configurations.

Station structure resembles a tree of nodes, of which there are different types defined by the `StationNode` objects assigned to each node. Lidgren.Network does not support the transmission of objects, only primitives, so it was necessary to serialize each station into a form that is more easily transmitted, like a string. This way it can transmitted to the server and relayed to connected clients, at which point it is de-serialized and built up into identical data structures on all clients. I decided that XML was a good way to represent the stations, since the tree structure of the stations can be preserved by nesting elements and the positions and types of the nodes can be defined as attributes.

The client therefore has two classes that deal with Station serialization and de-serialization, `StationToXML` and `StationFromXML`. `StationToXML` takes as input a `Station` object and recursively parses through the station tree, creating an element for each `Node` and nesting them as they appear in the station tree. For each node it also adds as attributes the node's UID (used to identify the node required by REMOVE_NODE messages), and the position of the node in the 2D space 'grid' so that it can be rebuilt by `StationFromXML`. An example XML representing a station can be seen in Appendix 14.17. Once the XML has been constructed for the station a CREATE_STATION message is constructed that contains the XML string with a unique identifier for the station. It is sent to the server for synchronization.

The server keeps track of the current stations and their state by storing them using the .NET XML API as `XmlDocuments`. The benefit of this is that whenever a node is destroyed by a player, the API has methods to remove elements of a XML document in memory. This is used to remove nodes that are destroyed to maintain an updated version of the Station states. Any players joining the server receive and receive all the Stations in the current game including their specific structure if they are partially destroyed.

The previous implementation of station synchronization involved the use of random seeds. Since stations are generated using a single `Random` object that is used to generated the tree and ultimately the 2D layout, providing the `Random` type with the same seed during instantiation results in the generation of a completely identical station. While sending a single int (the seed) instead of a fairly large XML string is superior from a bandwidth perspective to synchronize stations, it was only effective if all the players join at the same time. If a player joins later, after a station is partially destroyed, the seed would produce the original Station before nodes were destroyed, and since the server wasn't keeping track of the structure of the station, was impossible to relay which nodes had been destroyed.

10 Testing

10.1 Approach

The client and the server were developed mostly utilizing test-driven development. Throughout the software development process, small sections of code were written to achieve atomic functionality, and the software was then built and ran to see if it produced the desired outcome. If, not the code would be modified and tested until it achieved what I required, at which point I would move onto the next method or feature. For the most part, this approach worked well, by having an idea in my mind of how the software should behave, and fairly comprehensive requirements, I was able to test the software thoroughly by simply playing the game, and at some points, deliberately attempting to produce errors by interacting with the game logic in abnormal fashion.

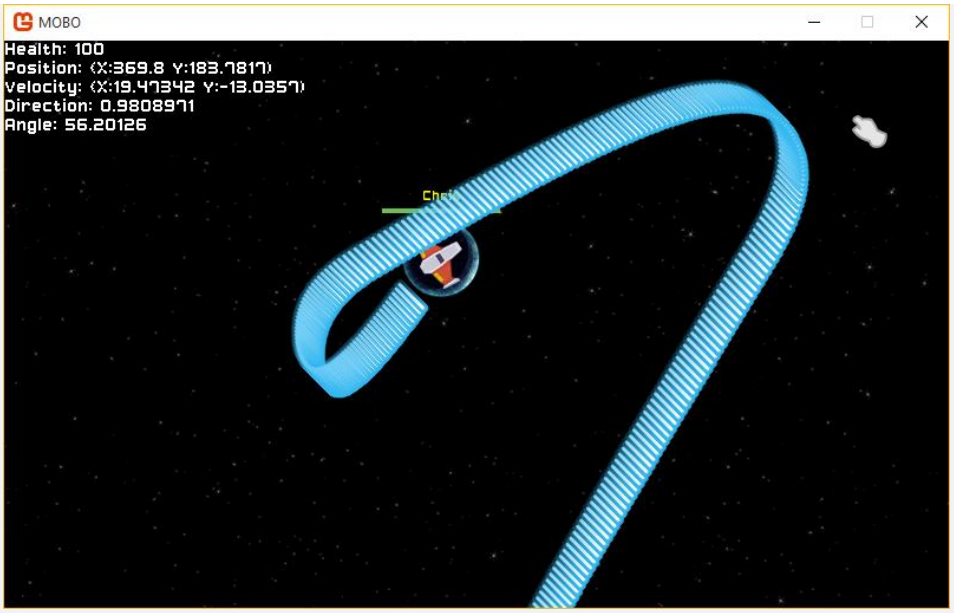

Figure 12 - An old screenshot of Mobo, showcasing some not-so-intended shooting behavior.

If errors or abnormalities did occur, it was a simple case of firstly identifying why the game logic was not behaving in the way I wanted it to, and then making a decision on how to fix the bug. Take the example in Figure 14. I had just written some code that would cause a projectile to be fired from the player in the direction they were facing if the space bar was pressed. Building and running the software showed that the projectiles were being fired 180 degrees in the wrong direction, the sprite was 90 degrees adjacent to the direction it should have been, and the player was able to shoot far too frequently. By returning to the code and adding Pi/2 to the resultant direction of the projectile, rotating the sprite 90 degrees and implementing a cool down timer between shots, I achieved the desired functionality. Most of the game data in Mobo has a visual manifestation and erroneous game state is often very noticeable, so I felt it was sufficient that rigorous play testing to ensure validity was effective for testing the software as opposed to unit testing individual classes.

10.2 Exceptions and stability

The vast majority of exceptions encountered throughout the development of the project were null pointer exceptions. Stations in particular were the most error prone. Previously the station generator was used to generate a tree, build a station in 2D space and then tree was used to recursively draw the StationNodes from the nodes in the tree. The problem with using the tree to draw the station is that the algorithm that built the tree in 2D space would often fail to place nodes in an appropriate space. As a result, for that particular node, there would be no `StationNode`. When the recursive draw method reached the node that failed to build, there would be no `StationNode` assigned to it, and an exception would occur. Using the Visual Studio debugger, I could see that certain nodes in the tree had null `StationNode` objects, and after coming up with multiple solutions, such as a method that would prune the tree of nodes that failed to build, the final solution was to use a flat list to draw the nodes that did not depend on the original station tree.

The majority of the testing I carried out was carried out by running the server and client on the same computer, for convenience. This was sufficient for testing the majority of the functionality of the game, but problems arose when the game was tested with a friend on separate computers over the internet. The game would crash in a seemingly random fashion, sometimes in just a few seconds of playing, other times after a few minutes. The culprit was the way I was storing collections of data, whether it be players, projectiles or stations. Oftentimes, the collections would be modified extremely frequently, especially while elements in the collections were being drawn or updated. Playing the game myself, I was unable to control two players at a time and test the concurrency. Switching the collection types that I had used to store collections to `ConcurrentDictionary` types from `List` on both the client and server has made the game very stable, and there have since been no concurrency issues since.

10.3 Stress testing

I employed stress testing to identify possible performance issues was stress testing. A previous implementation of representing the stations in 2D space utilized 100*100 2D arrays to store the stations. Clearly this was not ideal since a large majority of the array would remain empty once the station was built, resulting in inefficient memory usage. At the time I was still testing the station generation and I was only drawing one station on the screen, but when there were three or four the framerate of the game would be severely reduced. Mobo is designed to run at 60 Hz. If the `Update` method of the main type of the game takes too long to calculate, the `Draw` call misses its target of a frame draw every 16ms and the frame rate of the game drops. 2D arrays are very slow to loop through, especially ones with 10,000 cells. By giving each node a grid position as vector, explained in section 7.6, I was able to forgo using arrays, resulting in performance that can support many stations in the game without the client slowing down. During normal gameplay, there are never more than 3 stations, but it still showcases how performance efficient stations are now compared to their previous implementation. In addition, up to 20 clients can connect to a single server instance before there are noticeable performance issues, which is within the boundaries of requirement for a multiplayer game.

11 Critical appraisal

In hindsight, at the start of the project, I had relatively little scope of what the functionality of software was supposed to include beyond the original project proposal. I had an idea of what the final product should resemble, a multiplayer enabled game similar to XKobo, but as evidenced by the highly vague aims and objectives of the plan, and in particular the tasks in the Gantt chart, it is clear that I was mostly unsure of how the implementation was to proceed. Naturally as the project progressed, and I had researched the fundamental principles of network enabled games, the direction of implementation became much clearer.

Since the work schedule I had produced for the plan was poorly representative of the tasks and time it would take each task to complete, for the most part, I did not stick to the schedule. The Gantt chart gives the impression that by Semester 2 I would be mostly complete with the implementation of both the client and server, which was simply not the case. I had in fact completed a good proportion of the work by the prototype demonstration. The client was mostly feature complete in terms of the GUI, player movement and network code, and the server was mature enough that multiple clients could connect and see each other moving and shooting with some basic collision detection. I had not yet started on stations, which formed the vast majority of the work in Semester 2 but having a strong foundation which to improve and add to the game moving into Semester 2 ultimately eased the pressure off completing some of the more harder requirements.

Motivation starting the project was initially very high. Having previously programmed basic games in Java, I found programming with the XNA framework very intuitive and enjoyable. I was so motivated in fact, that ended up branding my game ‘Mobo’ (multiplayer XKobo), and spent a lot of time making logos for the main menu and designing the GUI to meet my own standards for video games. In hindsight this was probably time better spent programming, but it helped me get behind my project in a way I felt like I was developing the project as a hobby as opposed to just an academic piece of work. By later October the GUI and menu system was in place and fairly polished, and I moved onto getting the player drawn on the screen, accepting input, moving and shooting. I also found the Lidgren.Network library intuitive also, despite having to spend a long time designing how I was going to integrate it into the client and modelling the structure of the Network class.

I first experienced a drop in motivation after the interim report was submitted and my next task was to implement stations. I found it exceedingly difficult to visualize and model the data structure, growing increasingly frustrated. As a result, the month of December and early January was the most unproductive period of the project, with just 7 commits stretching over that period, compared to a total of 96 commits in the month of November. This ultimately reflects the pace at which work was completed during the early stage of the project; large bursts of motivation led to a lot of work being completed in short periods of time, but I also experienced periods a drop in drive, and in hindsight I could have been a bit more consistent and structured my development time a bit better.

In late January I began programming additional elements of the game, such as the mini-map, to ease myself back into development. Soon after I had begun work once again on the stations, and by mid-February I had basic but functional version of stations in place. At this point development advanced at a much more consistent pace. March was another very productive month, in which I further refined stations by optimizing their data structures (no more 2D arrays for storing nodes), fixing bugs, making them destructible, and writing the integral `StationToXML` and `StationFromXML` classes.

The final client is a large and complex piece of software at almost 1500 lines of functional code, and by far the biggest piece of software I have ever developed. While I have tried to program Mobo using best practices and programming techniques, some parts of the code base are either more bloated or more complicated than they need to be. An example of this is the StationNode class. StationNodes can be of three types, and all three have fairly different behavior, resulting in an unnecessarily long class with many conditions, and I could have instead created three separate classes that all inherit from a `StationNode` superclass, simplifying the code. There are a few more cases like this throughout the code, and I will definitely look to improve the quality of the code in the future as I continue developing the game beyond university.

Over the past decade there have been an increasing number of multiplayer-focused online video games that have failed to launch properly. While I acknowledge that Mobo is far simpler in network functionality than the average multi-million dollar budget multiplayer title, the project provided a very interesting insight into the general idea behind the communication protocols and messages that form the backbone of the synchronization that many of these games achieve. The design and implementation of the network systems and the servers that these games feature likely require a highly skilled, highly paid team of engineers and developers. It is for this reason that I think the basis for this project has never been more relevant nowadays, with huge demands on developers, not only in the video game industry, but when making any software that has significant network functionality.

It is for this reason that I also believe that the skills I learned throughout the project are not just applicable to game development but transferrable to any professional programming profession. Computer games can vary in complexity from the simplest text based games to the much larger pieces of software that are commonly developed by studios nowadays. These larger games, sometimes boasting hundreds of components and features, are significant programming challenges and often require large teams of software architects, developers and testers to be realized. This complexity arises from the nature of video games themselves; game mechanics commonly require complex data structures, algorithms, mathematical calculations and concurrent programming concepts, and online multiplayer games also requiring meticulous server and network protocol design and implementation. The project therefore provided not only a significant educational insight into many programming techniques used in video game development, many being mathematical and algorithmic in nature, but also due to being highly user-centric, presented an opportunity to learn about graphical user interfaces, user experience and visual design.

12 Conclusion

In general, I was fairly ambitious when setting requirements, particularly in the plan. As the project progressed, and possible paths of implementation became clearer, I shifted my aim from including all the features I had originally set out to implement and instead focused on polishing the more important core aspects of the project. As a result, seamless level looping was only achieved partially; all elements that can move are looped if they reach the boundary in the game state, but I did not manage to develop a performance-conscious method of visually wrapping the level. I understand the technical requirement behind achieving this, that the game environment should be re-drawn with the correct offset at the game environment boundaries, but I instead chose to focus my effort on the networking and stations as they were more important to the completeness of the final product, and were overall requirements with higher priority.

To conclude, I believe the product closely achieves all of the original aims and objectives of the project and a majority of the requirements have been satisfied fully. The end product is two part software consisting of a client and server, providing a stable and synchronous multiplayer experience. The client is performance efficient, as a result of frequent optimization throughout the project, and the server achieves full synchronization of the game state. Significant attention was paid to creating a game with graphically pleasing visual design and responsive user interface, as well as one that contains features of common video games such as a mini-map, user-configurable settings and collision detection. Both client and the server have been tested thoroughly, and employing stress testing has allowed me to see that the software performs well within the boundaries that one would expect of a game of this nature. The project provided valuable insight into the complex programming challenges encountered in game development, as well as the difficulty in maintaining a state of synchronization between clients that are concurrently interacting with the game state. I look forward to continuing the development of the Mobo and applying skills I have learned for creating new games and applications with network features.

13 Bibliography

1. Source Multiplayer Networking. *Valve Developer Community Wiki*, Accessed 15/04/2016, https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
2. High Level Networking Concepts. *Unity Manual*, Accessed 17/04/2016, http://docs.unity3d.com/Manual/net-HighLevelOverview.html
3. Lidgren Network Basics. *Lidgren Networking Library Google Code Wiki*, Accessed 01/05/2016, https://code.google.com/p/lidgren-network-gen3/wiki/Basics
4. Lecky-Thompson, Guy W. *Fundamentals of Network Game Development*. Herndon, VA, USA: Course Technology / Cengage Learning, 2008.
5. Rabin, Steve. *Introduction to Game Development (2nd Edition)*. Herndon, VA, USA: Course Technology / Cengage Learning, 2009.
6. Mount David. Varshney Amitabh. *Networking and Multiplayer Games*. Chapter 9 University of Maryland 2011.

14 Appendix

14.1 Architecture of XNA applications

The architecture of XNA/Monogame applications is constant across all implementations. XNA applications have four tiers of abstraction referred to as the XNA 'stack'. The highest tier consists of the application logic, essentially, all the code I have written for the implementation of the client and the game itself. My application runs atop the XNA Extended Framework which contains the application model. The application model provides the interface for the base game class of the client application. The XNA Extended Framework tier also contains a the content pipeline responsible for compiling and packaging assets into file formats that XNA can use more efficiently. With the introduction of MonoGame, content such as graphics and sound no longer has to be compiled into specific formats for the XNA class library to use and can instead be added to the project in their un-compiled forms using the MonoGame Pipeline tool, a separate application that replaces the XNA content pipeline.

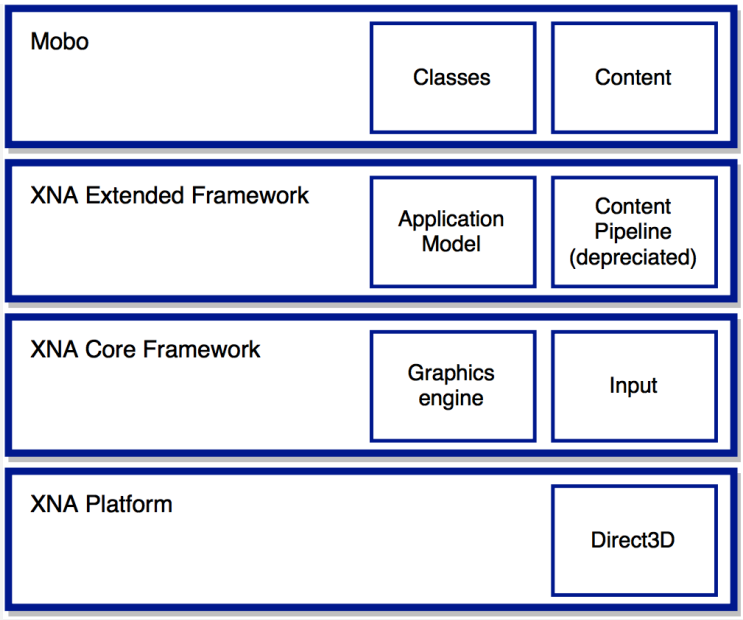

Figure 13 - The XNA 'stack', architecture of an XNA application.

The XNA Core Framework layer contains the graphics engine, responsible for providing methods to draw graphics, and the audio API to play audio assets. This layer also provides the methods used to access the current state of the keyboard and mouse, essential for capturing user input. The lowest-level layer, XNA Platform, runs atop .NET and is responsible for supporting the graphics engine by calling Direct3D's API and obtaining the input of the mouse and keyboard from the operating system. To summarize, only the top layer of the XNA 'stack' is provided by the developer, the extended and core framework layers provide APIs that the developer can use, and the final platform layer handles the low-level operations the layers above it require.

14.2 Architecture of Windows Presentation Foundation applications

The server is a much simpler application than the client and not a considerable step up from the original command line version if one disregards the GUI. The WPF application architecture resembles a simple one-way model-view-controller (MVC). The presentation GUI, defined by the layout XAML, including the player pane and the log text area acts as the 'view' and is updated frequently from the data structures of the model, namely the collection (`ConcurrentDictionary`) of players in the server. The controller is the main Program class of MoboServerWPF, which frequently invokes the Network class. The Network class is responsible for reading incoming messages, and updates the player list (the model) to reflect the new positions of players. Finally, the controller refreshes the view to display the new data. This refresh occurs every tick, so changes made to the model appear instantaneously.

14.3 XNA Game interface

```
namespace Mobo
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        public Game1() { }

        protected override void Initialize() { base.Initialize(); }

        protected override void LoadContent() { }

        protected override void UnloadContent() { } // Not required for Mobo

        protected override void Update(GameTime gameTime) { base.Update(gameTime); }

        protected override void Draw(GameTime gameTime) { base.Draw(gameTime); }
    }
}
```

Code 1 - A heavily simplified Game1.cs showing the basic structure of all XNA applications

14.4 LoadContent method of the XNA Game interface

```
protected override void LoadContent()
{
      // Initialise the spritebatches
      staticSpriteBatch = new SpriteBatch(GraphicsDevice);
      cameraSpriteBatch = new SpriteBatch(GraphicsDevice);

      // Load all the assets into memory (texures, sound etc.)
      new ContentStore(Content);

      // Initialise all the GameStates, loading their content
      mainMenu.Initialize();
      offline.Initialize();
      online.Initialize();
      help.Initialize();
      settings.Initialize();
      stationGenerator.Initialize();
}
```

Code 2 - Simplified LoadContent method of Game1.cs

14.5 Background class

```
Texture2D background;
Rectangle rectangle = new Rectangle(0, 0, 3600, 3600);

public void Draw(SpriteBatch spriteBatch)
{
      spriteBatch.Draw(background, new Vector2(-1800,-1800), rectangle,
                  Color.White, 0, Vector2.Zero, 1, SpriteEffects.None, 0);
}
```

Code 3 - Excerpt from Background.cs

14.6 Entity collections

```
public static ConcurrentDictionary<long, Player> players =
                                        new ConcurrentDictionary<long, Player>();
Background background;
Minimap minimap;
public static StationSpawner spawner;
public static Player localPlayer;
```

Code 4 - Excerpt from Online.cs showing some data structures of the in-game environment.

```
public ConcurrentDictionary<int, Station> stations =
                                        new ConcurrentDictionary<int, Station>();
```

Code 5 - Excerpt from StationSpawner.cs showing the data structure that contains the stations.

14.7 Mini-map

```
int height = 128;
int width = 128;
float range = 2000.0f;
int dot = 2;

private void DrawPlayers(ConcurrentDictionary<long, Player> players, SpriteBatch
spriteBatch)
{
	foreach (Player p in players.Values)
	{
		float dx = p.position.X - local.position.X;
		float dy = p.position.Y - local.position.Y;
		if ((dx > -range && dx < range) && (dy > -range && dy < range))
		{
			int gridx = (int)((dx + 2000f) / 4000f * 128f);
			int gridy = (int)((dy + 2000f) / 4000f * 128f);
			spriteBatch.Draw(ContentStore.debug,
				new Rectangle(map.Left + gridx - 1, gridy - 1, dot, dot),
				Color.Cyan);
		}
	}
}
```

Code 6 - Excerpt from Minimap.cs showing the code that draws players on the mini-map.

14.8 Rotation of the player sprite

```
public void Update() {
	...

	// Calculate the direction the ship is pointing based upon velocity
	playerRotation = calculateDirection(velocity);

	// Calculate the angle to the mouse from the screen center
	mouseRotation = angleToMouse() + MathHelper.PiOver2;

	...
}
```

Code 7 - Excerpt of Update method of Player, showing calculation of angles, Player.cs.

14.9 Trigonometric methods behind player rotation

```
public float calculateDirection(Vector2 velocity)
{
	if (KeyboardInput.keyPressed)
		return (float)Math.Atan2(velocity.Y, velocity.X) + MathHelper.PiOver2;
	else
		return mouseRotation;
}

public float angleToMouse()
{
	Vector2 mouseVector = ScreenManager.cursorPos - ScreenManager.screenCenter +
new Vector2(16, 16);
	return (float)Math.Atan2(mouseVector.Y, mouseVector.X);
}
```

Code 8 - The trigonometric methods themselves used for calculating angles from vectors, Player.cs.

14.10 Trigonometric method behind projectile velocity

```
// Calculate the starting velocity of a projectile based on an angle
public Vector2 projectileVelocity(float angle)
{
	float velocity_x = -(float)Math.Cos(angle + MathHelper.PiOver2) * bulletSpeed;
	float velocity_y = -(float)Math.Sin(angle + MathHelper.PiOver2) * bulletSpeed;

	Vector2 totalVelocity = new Vector2(velocity_x, velocity_y);
	return totalVelocity + velocity;
}
```

Code 9 - The method used to calculate the velocity of a new projectile in the direction the user is aiming, Player.cs.

14.11 Station tree structure

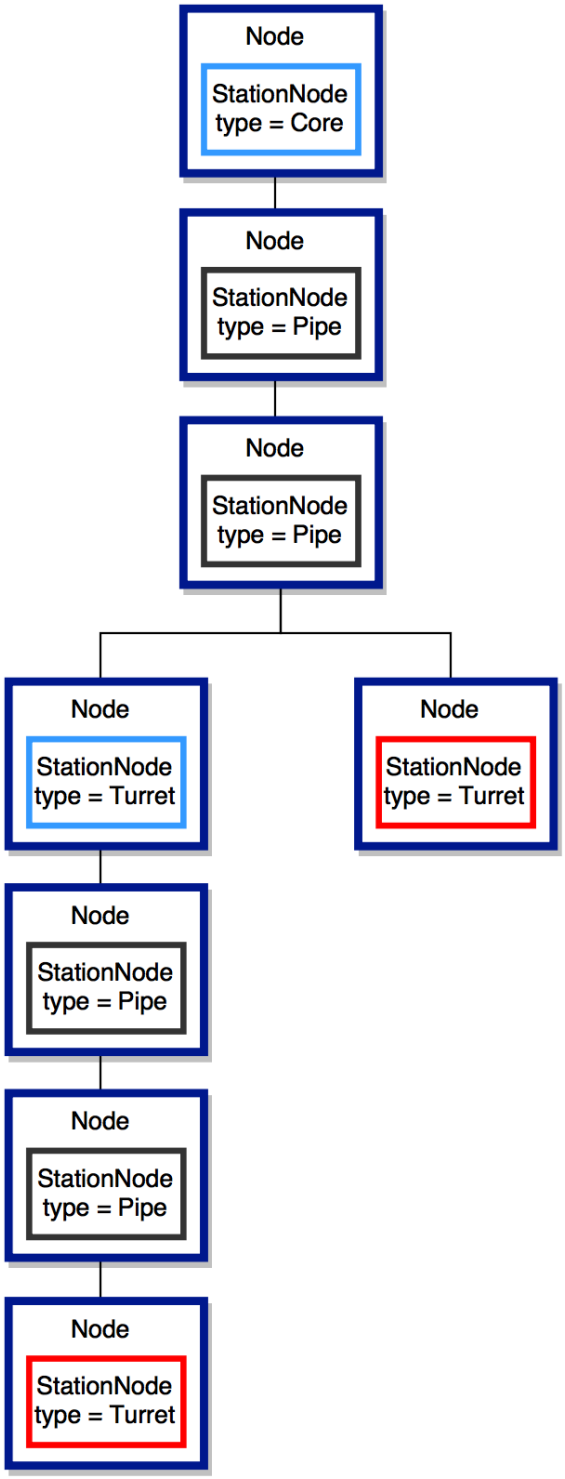

Figure 14 - A simplification of a Station tree.

14.12 Start of the station generator algorithm

```
public Station(Vector2 start_position, int max_depth, int difficulty, int
initial_branches, StationSpawner spawner)
{
	...
	// Initialise the station tree
	station_tree = new Node(0, null);
	station_tree.data = new StationNode(station_tree, this);
	station_tree.position = Vector2.Zero;
	vectors.Add(Vector2.Zero);
	...
}
```

Code 10 - Generating a station tree starting with the Core, Station.cs.

14.13 Generating an initial Node tree

```
private Node GenerateNodeChildren(Node node, int depth, int difficulty)
{
	int num_nodes = GenerateNumNodes(difficulty, depth);

	if (depth <= max_depth)
		{
			for (int i = 0; i < num_nodes; i++)
			{
				Node to_add = GenerateNodeChildren(new Node(depth, node),
					depth + 1, difficulty);
				node.Add(to_add);
			}
		}
	return node;
}
```

Code 11 - The method that builds the initial tree structure in Station.cs.

14.14 Building the station in 2D space

```
private void BuildStation(Node node)
{
	bool valid = false;

	HashSet<Vector2> poss = new HashSet<Vector2>();
	poss.Add(new Vector2(1, 0));
	poss.Add(new Vector2(-1, 0));
	poss.Add(new Vector2(0, 1));
	poss.Add(new Vector2(0, -1));
	List<Vector2> shuff = poss.OrderBy(x => rand.Next()).ToList();

	while (!valid && shuff.Count > 0)
	{
		Vector2 new_build_pos = node.parent.position;

		new_build_pos += shuff[0];

		if (!vectors.Contains(new_build_pos))
		{
			valid = true;
			node.position = new_build_pos;
			node.data = new StationNode(node, this);
			vectors.Add(new_build_pos);
		}
		shuff.RemoveAt(0);
	}

	foreach (Node child in node.children)
	{
		BuildStation(child);
	}
}
```

Code 12 - The method that builds the Station in 2D space from the station tree, Station.cs.

14.15 Message parse loop

```
// Read messages if they are not null and act according to the 'header' string
while((in_message = Server.ReadMessage()) != null)
{
	if (in_message.MessageType == NetIncomingMessageType.Data)
	{
		switch (in_message.ReadByte())
		{
			case CONNECT: Connect(); break;
			case MOVE: Move(); break;
			case DISCONNECT: Disconnect(); break;
			case CREATE_PROJECTILE: CreateProjectile(); break;
			case REMOVE_PROJECTILE: RemoveProjectile(); break;
			case CREATE_STATION_PROJECTILE: CreateStationProjectile(); break;
			case CREATE_STATION: CreateStation(); break;
			case REMOVE_STATION: RemoveStation(); break;
			case REMOVE_NODE: RemoveNode(); break;
			case HEALTH: Health(); break;
			case SCORE: Score(); break;
		}
	}
}
```

Code 13 - The message parse loop, from the Update method of Network.cs of the server.

14.16 Constructing and sending a message using Lidgren.Network

```
foreach(Player player in players.Values)
{
	...

	// Send positions of all players to all clients
	Network.out_message = Network.Server.CreateMessage();
	Network.out_message.Write(Network.MOVE);
	Network.out_message.Write(player.uid);
	Network.out_message.Write(player.x);
	Network.out_message.Write(player.y);
	Network.out_message.Write(player.rotation);

	Network.Server.SendMessage(
		Network.out_message,
		Network.Server.Connections,
		NetDeliveryMethod.Unreliable, // UDP!
		0
	);

	...
}
```

Code 14 - A message being constructed and sent using Lidgren.Network, from Player.cs of the server.

14.17 Serialized Station in XML

```
<Station id="300450198" X="720" Y="0">
  <Node id="375194367" depth="0" health="100" X="0" Y="0" type="Core">
    <Node id="625170993" depth="1" health="100" X="-1" Y="0" type="Pipe">
      <Node id="1892567263" depth="2" health="100" X="-1" Y="-1" type="Pipe">
        <Node id="1633194328" depth="3" health="100" X="0" Y="-1" type="Pipe">
          <Node id="1657699549" depth="4" health="100" X="0" Y="-2" type="Pipe">
            <Node id="1933598544" depth="5" health="100" X="0" Y="-3" type="Pipe">
              <Node id="1371948763" depth="6" health="100" X="0" Y="-4" type="Turret">
                <Node id="388807057" depth="7" health="100" X="-1" Y="-4" type="Pipe">
                  <Node id="794416869" depth="8" health="100" X="-2" Y="-4" type="Turret" />
                </Node>
              </Node>
              <Node id="1106763073" depth="6" health="100" X="-1" Y="-3" type="Turret">
                <Node id="787487506" depth="7" health="100" X="-1" Y="-2" type="Pipe">
                  <Node id="2048934173" depth="8" health="100" X="-2" Y="-2" type="Turret" />
                </Node>
              </Node>
            </Node>
          </Node>
        </Node>
      </Node>
    </Node>
    <Node id="1142469891" depth="1" health="100" X="1" Y="0" type="Pipe">
      <Node id="153703198" depth="2" health="100" X="2" Y="0" type="Pipe">
        <Node id="1084265335" depth="3" health="100" X="2" Y="-1" type="Pipe">
          <Node id="727170578" depth="4" health="100" X="2" Y="-2" type="Pipe">
            <Node id="945131465" depth="5" health="100" X="1" Y="-2" type="Pipe">
              <Node id="1760118062" depth="6" health="100" X="1" Y="-1" type="Turret" />
              <Node id="538806051" depth="6" health="100" X="1" Y="-3" type="Turret">
                <Node id="2141801052" depth="7" health="100" X="2" Y="-3" type="Pipe">
                  <Node id="497990408" depth="8" health="100" X="2" Y="-4" type="Turret" />
                </Node>
              </Node>
            </Node>
          </Node>
        </Node>
      </Node>
    </Node>
  </Node>
</Station>
```

Code 15 - An example of a Station serialized by StationToXML.cs.