- Tank Wars DS -

Implementing A Massively Multiplayer Online Game on the Nintendo DS

Daniel Jenkins[1]

March 2008

[1]email: daniel@danielj.co.uk

Abstract

This report is an investigation into a proof of concept for a Massively Multi-player game on the Nintendo DS; by connecting multiple clients to one main linux based server using homebrew software available for the Nintendo DS and C/C++. I do so by designing and developing server/client applications and implementing them successfully.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Nintendo DS is a popular hand-held gaming console, with touch screen interaction, and two extra buttons X and Y as well as the usual Game Boy style controls. It has support for wireless communication using standard Wifi protocol and Nintendo's own DS-to-DS wireless protocol (also known as Nifi). Although many of the games utilize these features in one form or another - be it multiplayer racing (Mario Kart) or online interaction (Animal Crossing) - as far as I am aware none of them currently support massively multiplayer gaming (several hundreds or thousands of players simultaneously).

After some thought on the matter, I could not reach any form of substantial conclusion that would prevent someone from creating and marketing a game with this feature in mind. I decided that this would be an excellent topic of research for my Innovations project, as not only is it an opportunity to create something that could directly influence the Nintendo DS games market, but it also compliments my original aim of using this project to improve my C/C++ programming knowledge and learn more about the process of software engineering. I was also intrigued by the challenge of designing and creating the program from initial concepts through to final product.

1.1 Aims and Objectives

My original aim for this research project was to ascertain whether or not the Nintendo DS is capable of running a MMORPG (Massively Multiplayer Online Role Playing Game). However, given the time frame that I had been allocated for the project - as well as my initial minimal knowledge of the Nintendo DS and its software/hardware capabilities - I decided it would be better to focus on a much more achievable short term goal.

I chose to focus on designing and creating the main structural aspects of an MMO (Massively Multiplayer Online) game and aim to create a basic proof of concept. I wanted to prove that a system similar in design to current solutions could be sucessfully applied using a PC server and Nintendo DS client. I felt that this was the most important feature to research as without this critical piece of architecture, there would be no way to build up any game concept around it. Without the support for large numbers of simultaneous users, the game would fail in that respect - be it a role playing, first person shooter, or real time strategy game.

In order to attempt to apply current-day standard designs of MMO games, I created a simple game concept to use as a basis for the design of the server and client modules. **Tank Wars** is a very basic top down 2D game concept where users load the game onto their DS, connect to and join a PC server, move a tank around the screen and try to shoot each other. With this in mind, I began to research into current MMO design solutions and how I could apply them to this games concept.

Chapter 2

Research and Development

2.1 MMORPG Networking Design and Structure

Suprisingly, MMORPG design and structure proved to be a larger topic of interest that I had originally expected. Clearly after the financial success of such games as Blizzards *World of Warcraft* [Blizzard, 2006], many people are keen to follow suit. Although I could not acquire any specific books on the subject, the internet and online **ATHENS** (Athens [2008]) resources proved to be very useful in obtaining research into that area. I managed to gather several *.pdf* documents, reports and journal extracts on the topic and began looking into MMORPG client/server design.

My main objective from the research was to work out a general formula for structuring an MMORPG server and client, by looking at as many examples as I could find. After reading through various papers it became apparent that there were several criteria by which the majority of designs were built to satisfy, shown in figure 2.1.

1. Manage multiple clients.
2. Employ distributable architecture and load partitioning.
3. Seperate world database
4. Efficient use of bandwidth

Figure 2.1: Criteria for MMORPG design.

With these criteria in mind I focused my research on obtaining a more technical / programming understanding of these points.

2.1.1 Managing multiple clients

In order for a game to be *Massively Multiplayer Online*, it needs to be able to support large numbers of concurrent users. My initial research and development required me to address this first point.

The Sockets API

I was already previously aware of a method of sending and receiving data over a network using the **Sockets API** and the C programming language. They allow the programmer to create a "socket" abstraction on a machine, associate it(or *bind* it) to an IP address, and then send and receive data through the sockets handle - which acts similarly to a file descriptor. I realised this was an ideal solution to base my client server architecture on, and researched further into network communication [Donahoo & Calvert, 2001, pg. 7].

TCP and UDP

There are two main protocol families available with the socket API, and they are **TCP** and **UDP**. Both offer end-to-end communication between server and client machines, however they vary in their design.

TCP(Transmission Control Protocol) sockets provide a *reliable byte-stream* method of communication. They create a fixed route of communication between a host and client machine designed to check for and deal with lost pack-

ets, duplicates and any other errors that might occur [Donahoo & Calvert, 2001, pg. 5]. Packets of data are "streamed" between one computer to another, and because of this it does not preserve message boundaries (it does not maintain the size of packets that it receives). In some respects this makes sending data over a network much easier, but due to the nature of network gaming this is not necessarily a good thing because of the restrictions created by a permanent connection. Using TCP sockets on a server machine is relatively straight forward. The socket is *created*, then *bound* to a particular IP on the machine (usually the one associated with the appropriate network interface). Then the socket is set to a *listen* state, where it blocks and awaits a connection request from a client machine. Once it receives that request, it then *accepts* the connection and creates a new temporary socket. Data can then be sent and received on the temporary socket until it is *closed*.

Client applications using TCP sockets follow a similar format. The socket is again created, but then a call to the *connect* function requests a connection to the server machine. Then the socket can be used to send and receive data from the server.

UDP(User Datagram Protocol) sockets are similar to TCP ones, however they uses a connectionless or *datagram* approach to communication. Unlike TCP sockets where the socket must be bound to an IP address - the socket is associated with an IP and a *port* number. Packets of data are treated more like seperate messages than a stream of data, and so message boundaries are preserved using UDP. There is also no real difference in using UDP sockets on a client or server application, both *create* a socket and then call *sendto* or *recvfrom* functions to send and receive data. The *sendto* function sends a packet of data to a specific IP address and a specific port number, and is not concerned whether or not the data arrives. This means that UDP provides no substantial error checking, which would have to be taken into account in the application's design.

The dsWifi(see Section 3.3.3) homebrew package (The term used online in

reference to custom made software for the Nintendo DS is **homebrew** *(See Hull [undated])*) turned out to contain a ported version of the sockets API, which was ideal. After some experimentation, I found that I could only get the UDP protocol style sockets working. This was not a problem however as I believe UDP sockets to be the better choice for an MMO client/server application. The need to send updates regularly means that occasionally losing a packet of data becomes trivial (unless of course this occurs frequently). TCP sockets are more reliable in terms of sending data but because they require to be directly connected to the client they lack the functionality of UDP sockets; where one socket can send and receive to as many clients the programmer wants to. TCP packets also have considerably larger headers than UDP ones (72bytes and 28bytes respectively)[Bogojevic & Kazemzadeh, 2003, pg. 22] As MMO and MMORPG games in particular apply certain bandwidth saving techniques, this freedom becomes very useful (as described in Section 2.3)

2.1.2 Distributable Architecture and Load Partitioning

As stated earlier; in order to be *Massively Multiplayer*, a server must be able to handle very large numbers of concurrent users. However, there will always be a finite limit to how many one server machine can handle; be it due to memory or processing power limitations. An important feature of MMO architecture is the ability to distribute the load over several CPUS. The most obvious way of doing this involves modulating the core elements of the server so that they can be run on different machines.

Figure 2.2 is based on the MMORPG server-side architecture design diagram as shown in [Caltagirone et al., 2002, pg. 108].

This design example clearly shows the possibility of modular distribution of the server. There is a central **Governer** module that controls several other modules, such as the **World** and **User** modules - all of which could easily be implemented on different machines.

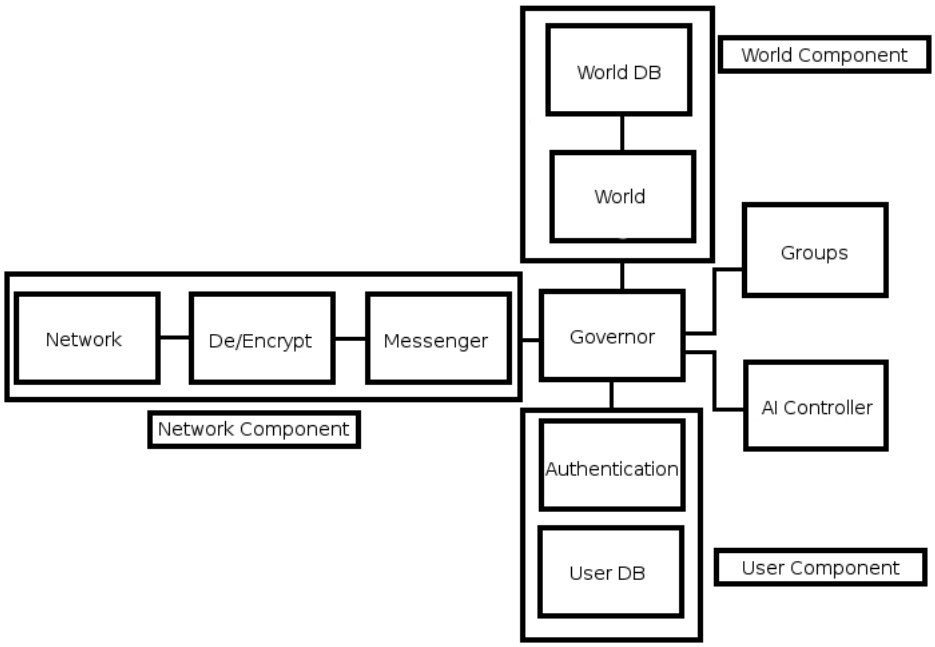

Figure 2.2: Server side architecture

2.1.3 C / C++

Due to the nature of C++'s object oriented design ethos, it makes it a perfect language for writing my client and server programs. I can create classes to deal with the object abstractions created by several users, and then use these structures or classes in my communication protocol. It also seemed to be the only logical choice of language as all of the Nintendo DS homebrew I found was written to support C or C++.

2.2 Seperate world database

Falling under a similar requirement as section 2.1.2, using a sepearate world database is more suited to MMORPG designs, where the world that the user interacts with is usually extremely large. By keeping the world database seperate to the main server, it prevents the server from becoming overwhelmed with database requests whilst simultaneously trying to handle other client requests.

A lot of time and work can be saved by using pre-existing databasing packages such as MySQL, which operate as a seperate entity to the server application. The server can query or set data within the database whilst leaving the main work to the machine running the database [Caltagirone et al., 2002, pg. 111].

2.3 Efficient use of bandwidth

There are several proposed solutions to the problems of bandwidth and latency [Bernier, 2005]. These vary depending on which specific type of MMO game is being made. However there are a few techniques that can be applied to most scenarios which are outlined as follows:

2.3.1 Protocol Design

A protocol is the process of encoding the data that is sent between two computers. By figuring out exactly what information needs to be sent (e.g function requests, location data etc.) you can refine the data more efficiently. Consider the following C example:

```
typedef struct
{
int posX;
int posY;
unsigned short ID;
char functionCall[4];
} protocol;
```

Figure 2.3: An example of data to be sent between computers

Here, the protocol consists of two integer numbers(4 bytes each), a string of 4 bytes and a short number(2 bytes). The first thing to consider is the possibility of padding. Most compilers, unless instructed otherwise, will *pad* this structure with 2 extra bytes, resulting in a total size of 16 bytes instead of 14 bytes. This is because "the compiler aligns the fields of structures to certain word boundaries based on their type." [Donahoo & Calvert, 2001, pg. 30,31].

The compiler makes the data structure maximally aligned so that the memory addresses of the components are divisible by the largest native integer (in this case, 4 bytes). It does so by adding 2 bytes of padding after the **unsigned short** making it upto 4 bytes in size.

From a protocol design point of view, sending empty bytes is not exactly efficient. This problem can be solved by either ordering the structure so that each memory address is divisible by the largest integer, or specifying to the compiler not to pad the structure. In C using *gcc* this is done by adding the **__attribute__ ((packed))** compiler attribute after the structure definition.

Packet compression is also a common technique. Using the example in Figure 2.3, say that there were a maximum of four different function calls available - move, look, kill and die. Instead of sending these as literal strings, you could replace the **string** of 4 bytes with a **char** of 1 byte, and use the first four individual bits as flags to specify which function to call (this would be decoded again at the server). Applying this technique already creates a reduction of 3 bytes. There are also more specific algorithms available to deal with the compression of data in general that can be applied to data packets.

There are several other techniques to reduce bandwidth, including **packet aggregation** which involves merging together several packets into one [Smed et al., 2002, pg. 3].

2.3.2 Interest Management

Sending information to each client regarding every other client on the server would soon lead to latency problems. It would be far more efficient to only send information to clients about other clients that were in their area of interest, therefore greatly reducing the number of packets sent and reducing the total bandwidth of the server. This technique can be executed at varying levels of depth. At the highest level, it can simply involve grouping clients depending on their location within the map - assigning different groups to different areas. At a deeper level of complexity, each client could have a *focus* and a *nimbus*.

The focus being the clients visible presence and the nimbus representing their field of view. This would allow for asynchronous message filtering where one user might be aware of the second user, whereas the second user might not be aware of the first user at the same time [Smed et al., 2002, pg. 6],.

2.3.3 Dead Reconing

Another technique to reduce bandwidth is to lower the frequency of packets sent to the clients. However doing so reduces the amount of information the client is given. **Dead reconing** is the method of applying client-side prediction in order to minimise the effect caused by the lack of information. For example; if a client was only being updated with data every 0.5 seconds, and was engaging in a snowball fight with another player, the position of the objects(in this case the snowballs) would only be updating every 0.5 seconds. This would cause visual problems whilst drawing the path of the objects - they would appear to be warping and jerking accross the screen. However if the clients used the initial information to predict the path of the object inbetween updates, then the visual errors would be minimized. This can be extended even further by using a convergence algorithm to smooth the transition between the predicted path and the actual positions of the object [Bogojevic & Kazemzadeh, 2003, pg. 25],[Smed et al., 2002, pg. 7-9].

Chapter 3

Implementation

3.1 Software Designs

The following are some software visualisations of my client and server designs. I used these to help me plan the best way of approaching certain problems, and also to help me bring my research together in a visible form.

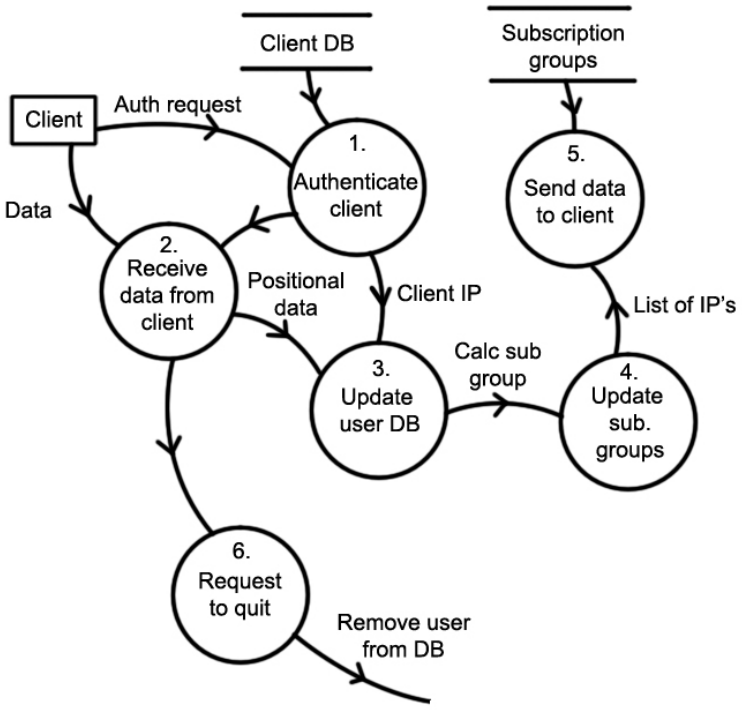

Figure 3.1: Server dataflow diagram.

This demonstrates the flow of information around my server design solution. This is the product of my research and practial application.

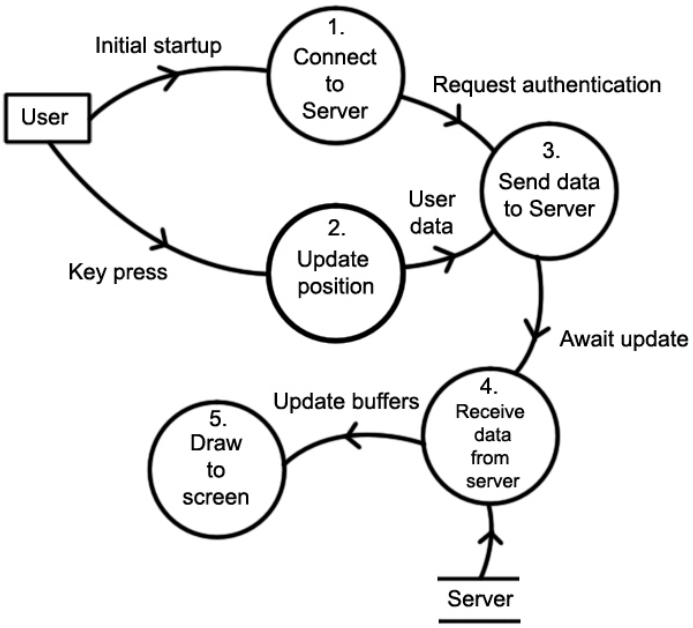

Figure 3.2: Client dataflow diagram.

This demonstrates the flow of information around my client program. Admittedly I did not focus as heavily on my client design as I did on the server, as I did not feel that I knew enough about the technical aspects of the Nintendo DS to readily offer architectural solutions. This is definitely something I would have liked to have focused on more had I set aside time to do so.

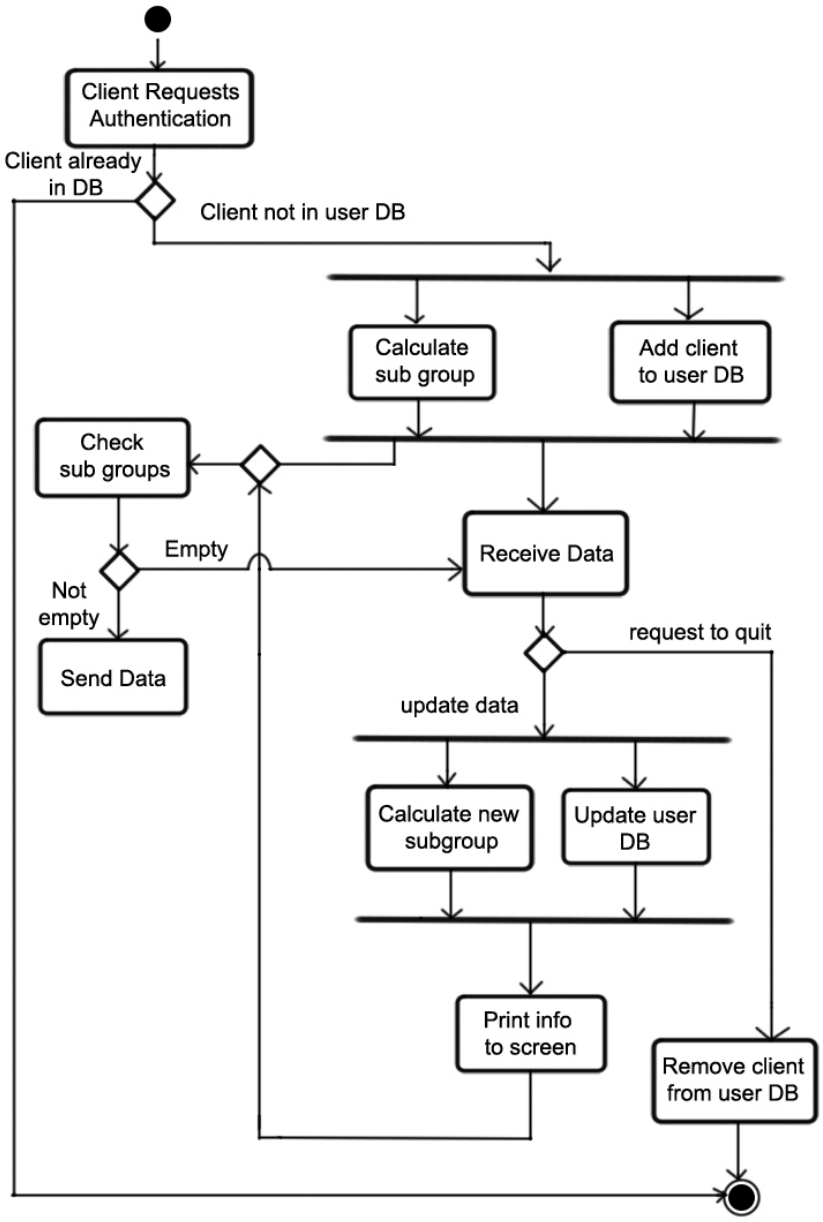

Figure 3.3: Initial server UML design.

A simple UML(Unified Modelling Language) diagram showing the process of handling clients on my server.

3.2 The Server - Ubuntu Linux PC

I considered the server to be the more important of the two parts of my software development, and so I decided to invest more time into applying as many features from my research into it as I could. Having not done much software design in the past, I was somewhat unsure of how to approach constructing the server. I decided that the best way to start would be to figure out what features I would need in order to satisfy as many of the criteria for a MMO server as possible (see Figure 2.1). I decided to divide my server code into modules (seperate .cpp files) that could be compiled seperately and then linked together to form the main executable. Not only would this be good practice in creating a well structured project, but it also satisfies the second criteria I state in Figure 2.1. Each module could easily be extended into a stand-alone executable which could run seperately on another machine.

Server machine specifications:

- AMD Athlon XP 2600+ 2.08GHz
- 512MB RAM
- Ethernet LAN Connection
- nVidia GeForce 4 Ti 4400 128MB Graphics Card
- Operating System: Ubuntu 6.06

I divided the modules into what I believed to be the main functional aspects of a MMO server. They are as follows:

3.2.1 Communication Module (twComm.cpp)

The communication module is one of the most important modules in the server. It handles communication between the client and the server, and initiates the socket abstractions in order for the client to be able to communicate with the server. Another important feature of this module is that it also initiates two

seperate *threads*, one which deals with client authentication and a second that deals with receiving data from the clients. These threads are required due to the main properties of the recv() and recvfrom() functions included in the sockets API. By default both the functions *recv()* and *recvfrom()* block whilst awaiting to receive data. This means that the thread of execution waits at these functions until data is received. This would not be at all practical in use within a server application, as it needs to handle several clients at once whilst still performing other operations.

Using threads however does incur a few more considerations. I needed to find a way of sharing data between the thread and the main process. This can be done easily enough, as by nature threads have access to any variables present in the scope they were created in. However, signaling to the process that there was new data to be accessed and then only allowing one thread to access that data at a time, was a challenging problem. After a little research and some advice from course lecturers, I discovered that the POSIX threads that I was using had some *mutexing* functions available. These essentially are locking devices which ensure that only one function is allowed access to shared data at any time. It involves *locking* the mutex before a function accesses it, and then unlocking it afterwards. If a second function has already locked this mutex, the call to the lock function will block until the mutex has been unlocked. This gave me a way of ensuring that only the process or the thread was accessing the shared variables at any one time.

As I was using UDP sockets - which allow many clients to send data packets to one socket on the server - I only needed to use one extra thread of execution to deal with receiving data. If I was to use TCP sockets which require a dedicated connection between the server socket and the client, I would have needed to either create a new thread or "forked" the process for every new client. This would have created a much larger overhead on the server and would have greatly reduced the maximum number of clients that could be connected at any one time. It would also have been a debugging nightmare

trying to keep track of so many threads of execution.

Table 3.1: twComm.h Main Functions.

Function	Description
int twSetupSockets(string);	Sets up sockets for send, recv, and auth
void *twRunAuthThread(void*);	Runs thread listening for connections
void *twRunRecvThread(void*);	Runs thread for outputting data to clients
int twSendACK(struct sockaddr_in*, int);	Sends DS its userID
int twSendData();	Sends data to designated client

```
typedef struct{

int X;
int Y;
int userID;

}twProtocol;
```

Figure 3.4: twProtocol.h: Protocol used by the client and server.

3.2.2 Subscription Manager (twSubscript.cpp)

This module is the subscription manager of the server. I decided to follow a design similar to a *Producer/Consumer* [Eisenhauer & Schwan, 2006] model for my server, where clients would subscribe to receiving information that they were interested in. In order to keep it simple for my proof of concept design, I limited the subscription groups into 4, each representing one of four quadrants of a map area. I took the map to be a total size of 256x256 pixels large, and so divided the map into 4 regions each with an area of 128x128 pixels.

The subscription manager currently uses 4 container classes, which in turn store the userID of each client associated with that subscription group. When a user first connects to the server, it has a position of (0,0), and so gets added to the appropriate subscription group (Group 1).

I could have implemented this decision making aspect on the client machine, but I decided to let the server calculate which subscription group a client should belong to. This was because I wanted the server to handle as

much of the processing as it could, in order to alleviate some strain on the much more limited DS's processing power, and hopefully improve the response times of the client.

Table 3.2: twSubscript.h Main Functions.

Function	Description
void twInitSubGroups();	Initialises sub groups.
int twAddToGroup(int, int);	Add a user to a sub group.
void twUpdateAllGroups();	Updates all users within each group.
int twCalcGroup(twUser* user);	Calcluate sub group for user
int twUpdateSubGrp(int, twProtocol*);	Updates subgroup

3.2.3 Database Module (twDatabase.cpp)

This module contains the main databasing functions of the server. Although I did not manage to apply any kind of "seperate world" database (I did look into using MySQL, though learning it was not feasable in the timeframe available), it still stores the main client database (userDB). This is done again using a vector container class containing instances of the **twUser** class (detailed in Figure 3.5). It stores each client's IP/connection information in a *sockaddr_in* stucture, and assigns each user with a **userID** number. These details are then used by the communication module (Section 3.2.1) and the subscription manager (Section 3.2.2). I did not have time to implement some of the following functions, because I deemed them as lower priority that the others.

Table 3.3: twDatabase.h Main Functions.

Function	Description
void twOpenDBFile();	Opens database file to store user info
void twWriteDBFile();	Writes data for user into DBFile
void twInitUserDB(void);	Starts linked list of user objects
int twCheckClientIP(struct sockaddr_in);	returns 0 if client is in DB, 1 if not
void twAddUser(struct sockaddr_in);	Add user to vector list and to sub group
int twRemoveUser(int);	Remove user from vector.
void twUpdateUser(userID, data);	Updates user details(x,y, grp etc.)
void twCloseUserDB();	Closes and frees linked list

```
class twUser
{
private:
int userID;
struct sockaddr_in sockInfo;
int posX;
int posY;
int subGroup;

public:
twUser();
~twUser();
int getX(void);
int getY(void);
int getID(void);
void setID(int);
void setX(int X);
void setY(int Y);

void getSockInfo(struct sockaddr_in );
void setSockInfo(struct sockaddr_in );
}
```

Figure 3.5: The twUser class definition

3.2.4 Display Module (twDisplay.cpp)

This module is not necessarily an important module, however its main purpose was to help with debugging whilst I wrote the other modules. Its main job is to print out information regarding clients that are connected to the server. I extended this further and also had a visual representation of the game map simulated using OpenGL.

Table 3.4: twDisplay.h Main Functions.

Function	Description
void twDisplayTotalUsers(void);	Prints out total users connected to server
void twDisplayAllUsers(void);	Prints out info about connected users to screen
void twDisplayUserInfo(int ID);	Prints out info about a specific user

3.2.5 Artificial Intelligence Module

If I had more time available for this project I would very much liked to have implemented this module, however I thought it to be of a lower priority than

the other more critical modules. This module would have contained all the functions and code relating to any artificial intelligence for *Tank Wars*, which would have been ideally executed on a seperate machine.

3.3 The Client - Nintendo DS Lite

Figure 3.6: A Nintendo DS Lite

3.3.1 Short Overview of Nintendo DS Hardware

The Nintendo DS Lite uses two **ARM** (Advanced RISC Machine) processors, the **ARM7**(33MHz) and the **ARM9**(66MHz). The ARM7 handles the user input and part of the 2D graphics core. It is also the only processor that can access the microphone, sound output, touchscreen, wifi communications and the real time clock [Anonymous, 2008]. I managed to find a standard ARM7 template file that I could use for my project - from which I made a few modifications.

The ARM9 processor is designed to do most of the "hard work". The majority of the application code is run on this processor, and it takes care of the 3D rendering core and also the 2D graphics core.

Aside from RAM caches available to the ARM9 and ARM7, the DS only has approximately 4MB of working RAM available. More memory can be utilized through the flash card with homebrew versions of DLDI(Dynamically Linked Device Interface) and *libfat*, but otherwise the executable size (total for ARM7 and ARM9) must not exceed this limit.

Having two processors running in parallel certainly has its benefits, however it also creates an interesting challenge for software design. There are a few mechanisms for IPC(Inter-process communication) but I found them to be quite tricky to implement - and so have not included many in my final code.

Although the DS supports Wifi, it is limited to approximately **2Mb/s**. This means that time and thought must be spent in designing an efficient protocol and applying other bandwidth saving techniques to keep the bandwidth and packet size to a minimum. Otherwise there is a strong risk of a bottleneck occuring on the clients side due to latency issues.

I decided to use my laptop to host my DS programming environment, because that then left my PC free to work on my server code simultaneously. My laptop specifications are as follows:

- Acer Aspire 1800

- Pentium 4 3.2GHz with H/T technology
- 512MB RAM
- Ethernet LAN connection
- ATI Mobility Radeon X600 128MB
- Operating System: Ubuntu 6.06

3.3.2 Supercard DS One

In order to even consider using the Nintendo DS as a main part of my project, I first had to ensure that there was sufficient support for creating home-made software on the DS. There are many different flash cards and software packs available online for the DS to run custom made homebrew *.nds* programs, and after some research I decided to purchase a **Super Card DS One** *(See Supercard [2006])* as it claimed to have very good support for homebrew software.

3.3.3 Nintendo DS Homebrew

Once I had a way to run homebrew software for the DS, I then had to develop a programming environment to work within, and to look into existing software and tools available to create and compile the code. In order to improve the efficiency of my search, I used the following criteria whilst browsing online:

1. I wanted to be able to avoid doing as much lower-level hardware programming as I could, as although this might be a substantially more efficient way of working, the benefits of doing so did not outweigh the time I would have needed to invest into learning it.
2. Compatibility with Ubuntu linux operating system.
3. I intended on using C/C++ to develop the software, so support for these languages would be ideal.
4. A way to interface with the Wifi features on the Nintendo DS.

5. General tools to handle compiling my code into a Nintendo DS executable.

Out of my research online, I soon discovered a reasonably large (if not slightly outdated) fanbase for homebrew application for the Nintendo DS. Thankfully I found a few programs and libraries that would help me develop my application.

devkitPro

The main tool-kit for developing homebrew applications on the Nintendo DS (and also other consoles that use the same processor architecture - such as the Game Boy Advanced and the Gamecube) is called **devkitPro** and is available freely at http://www.devkitpro.org/. This provides the ability to cross compile C/C++ code written for the DS and creates an executable that can be copied onto and executed on the DS. It runs on linux and windows, which makes it ideal for this project. It also comes with some template makefiles which makes structuring the project files simpler and compiling very easy (once a few global ENV variables are setup).

libnds

Included in the devkitPro toolkit chain is a library written by Michael Noland and Jason Rogers, which provides an intermediary abstract layer between the programmer and the Nintendo DS hardware. It allows the programmer to access the majority of the Nintendo DS's hardware features via global *#define's* and C functions, which means (with the help of the source files documentation and online tutorials) that it not too difficult to get to grips with the basics of the Nintendo DS core features and functions.

dsWifi

Written by Stephen Stair, **dsWifi** is a module that utilizes the Nintendo DS's wireless features. Although it has recently been added into the devkitPro

toolkit, I had some difficulty implementing it into my programming environment. I managed to resolve this problem however and reinstalled a different version with no problems. This library of functions and global *#define's* allows you to access the wireless features of the DS via function calls. It also accesses the networking information stored on the Nintendo DS. As several games on the DS utilize the wireless feature, the DS has a small built in application that allows you to store the connection information to 3 different access points (which it retains until changed). This means that once the details for the wireless access point you are connecting to have been stored in memory, you can access this data using dsWifi functions - making connecting to a wireless network very easy.The dsWifi package also contains a ported version of the *sockets API*(see Section 2.1.1).

3.3.4 Development

Getting to grips with Nintendo DS programming took a little while. I spent quite a lot of time looking at available tutorials online; some of which turned out to be outdated and did not quite work properly with the newer release of *devkitPro*(Section 3.3.3) that I was using.

I started my development by asessing what the key features I needed to implement, and prioritised my development accordingly. My first task was to see if I could get a simple "Hello World" program to run, and from that try to understand the inner workings of the program structure and function calls. Whilst doing so I spent some time reading through the documentation for *libnds*(Section 3.3.3) which can be found in the individual source files. I found that some features on libnds did not work in conjuction with some of the template code I found online. I then had to spend a lot of time working through the libnds header files and figuring out which functions/variables were missing or had been renamed, and working around them.

Once I had managed to get "Hello World" running, the next key feature I needed was interfacing with the Wifi. Again it took longer than anticipated to

get this feature working, and in the end relied on using template code written by *Stephen Stair*(see Section 3.3.3). However with the Wifi working I could begin development of my server code at the same time.

3.3.5 Basic Features

Users copy the *.nds* compiled executable onto the DS. When it is run, it attempts to connect to the server PC (via IP) by sending an authentication request to the server. Once the server approves this request, it sends an acknowledgement packet. The user can then change their position using the D-Pad, updating their position on the screen. The client sends data to the server, and then waits to receive data before checking for user input once more. It then sends the positional data to the server and again awaits an update from the server before continuing.

Unfortunately, I did not manage to implement any advanced graphics on my *Tank Wars* client, although I did spend a lot of time trying to display graphics using the 2D core features on the DS. I managed to display a tiled background using a simple tile, tilebase and mapbase (See [Vijn, 2007]) but realised that this was an unnecessary component to proving my concept. If I had more time I would have liked to work more on the graphics. It is definitely something I will work on extending in the future, and hopefully should be able to get some simple sprites representing the tanks themselves.

Figure 3.7: The DS client connected to the server.

Chapter 4

Conclusions

At the end of this project I had:

- Connected my DS to my linux PC and used it to drive an 'X' representing the client around the screen.
- Used available homebrew APIs to use the features of the Nintendo DS using C and C++.
- Created a simple graphical representation of the user's "Tank" on the top screen of the Nintendo DS (see Figure 3.7).
- Created an authorising system where the DS requests to be added to the server database, and the server returns an ID for the client.
- Employed a protocol for communication between the DS and the server.
- Written a server using C/C++, threads and the sockets API to accept and handle multiple clients.
- Used openGL to create a visual representation of clients connected to the server (see Figure 4.2).
- Used C++ to create an object abstraction of a client for use in a dynamic database.

- Used the object orientation features of C++ to store each client as an object in a linked list (container class) on the server.
- Created a subscription group system to handle users within similar areas of a map space as a technique to save bandwidth.
- Successfully organised my server code into individual .cpp files for ease of use and expandability.

After spending a lot of time time getting to know the capabilities of the Nintendo DS and constructing my client/server programs, I have gained a considerable amount of knowledge in network programming and C/C++. Although several aspects of development took longer than anticipated, I am by no means disappointed with the outcome. I am grateful that I decided to simplify my initial aims, as originally they were much too ambitious in the time frame available. I have however, sucessfully managed to prove that the Nintendo DS is capable of connecting to a Linux based PC, and has the capabilities to interact with other users on that machine. I feel that this is a solid foundation to build upon if I wanted to continue working towards creating a *Massively Multiplayer Online* game on the DS. There are several things I would have liked to have added to my final server program which I feel are important features; such as an AI module and encryption of data packets. Likewise I am a little dissapointed that I did not have time to execute my simple games concept of *Tank Wars* on the DS, as I think that its simplicity could make it very successful. Also I would have liked more time to develop the Nintendo DS client to receive information from the server about the people in its subscription group.

I am particularly happy with my graphical representation of the users on my server - displayed via openGL. It was very enjoyable to connect several Nintendo DS consoles successfully to a server that I had written and then see the results directly on the screen. I was also impressed that I had overcome all of the apparent problems I had encountered whilst designing and executing

my project. I was also very happy with my implementation of threading as my solution to manage multiple clients. I had not had any previous experience with threads or mutexing, and I was happy with the result. I am very pleased with both my server and client programs, and my only regret was not having enough time to advance them further.

I feel that the client program didn't go as well as I would have hoped. However, if I had been working on a PC to PC client/server application, then I would have made much more progress. I found it hard at times trying to figure out the functions and workings of libnds, and dsWifi. It seemed that half the battle was trying to get the programs to work in my programming environment. This wouldn't have been such an issue had I been building my client on a linux PC, although I would not have learnt half as much as I have. The support available online via tutorials was somewhat limited, but I did get some help from online forums. Although I did manage to implement simple graphics on the Nintendo DS via its framebuffer mode, it is an area I would have liked more time to develop.

Although my client/server system is not necessarily *Massively Multiplayer*, it demonstrates that more players could be possible with more resources. By modulating the components of the server and abstracting away the world database, I believe that a DS MMO is very possible. Running the world database and AI modules on seperate or multiple machines would be ideal. It is important however to consider the limitations of the DS, as this is the only part of the design which is restricted in that sense. Packets of data must be well encoded to coincide with the 2MB/s limit of the DS, and the executable itself should not exceed 4MB. If I had the resources and the time available I would have liked to have developed a more hardware driven design solution, as I'm sure that emulating a sockets API is not the most effective solution.

I did not realise how important it is to design and visualise an application before beginning practical work. This is something I personally would like to improve on, as I felt that it was the hardest aspect of my project - and I

only just touched the surface of the sofware design process. I will make sure that my aims in any future programming projects include a more in-depth research into the design process. Overall the whole experience has not put me off continuing this project on my own time, and I hope that a project like this will be of interest to other games console software developers.

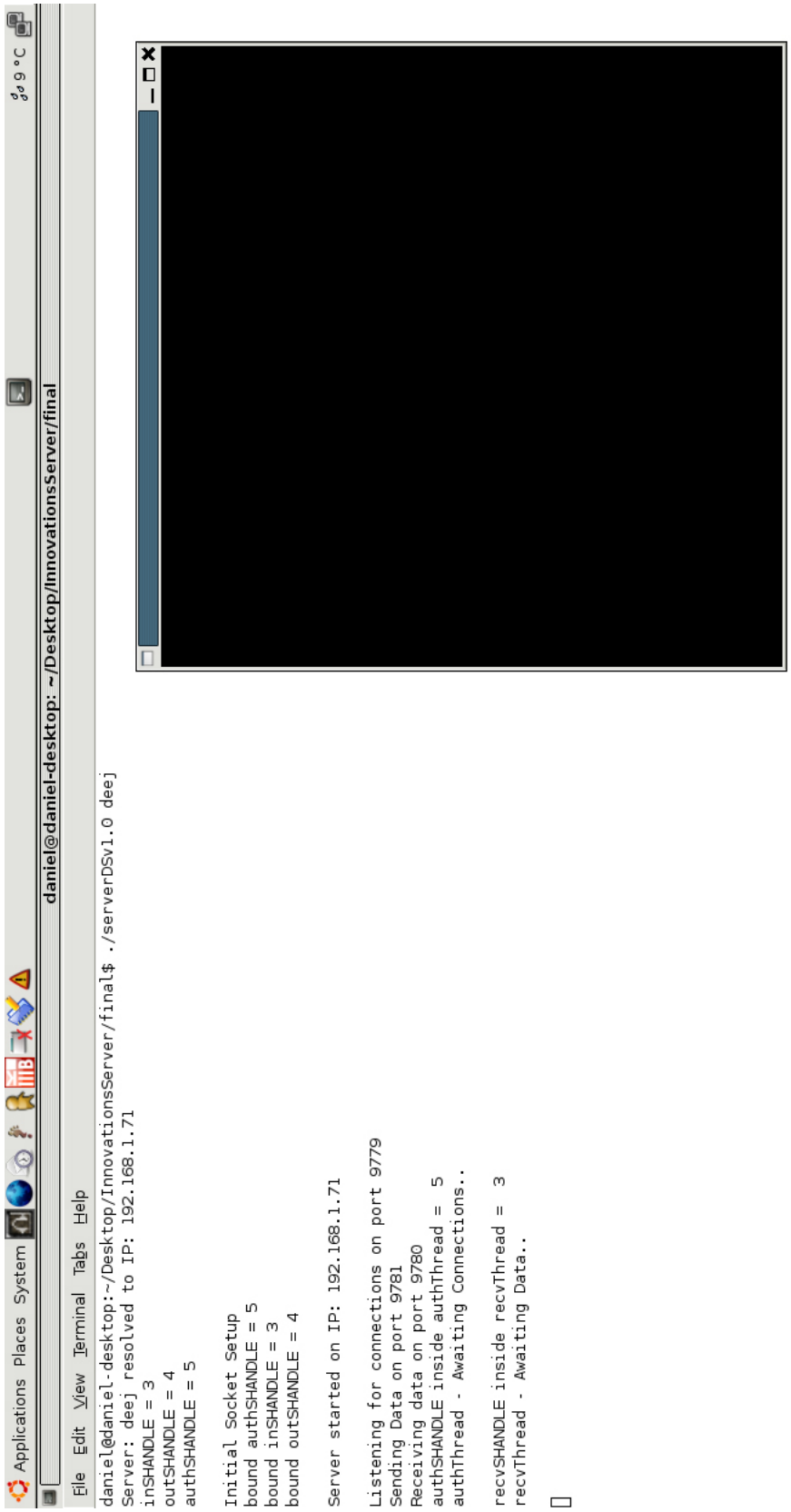

Figure 4.1: The server awaiting connections from clients.

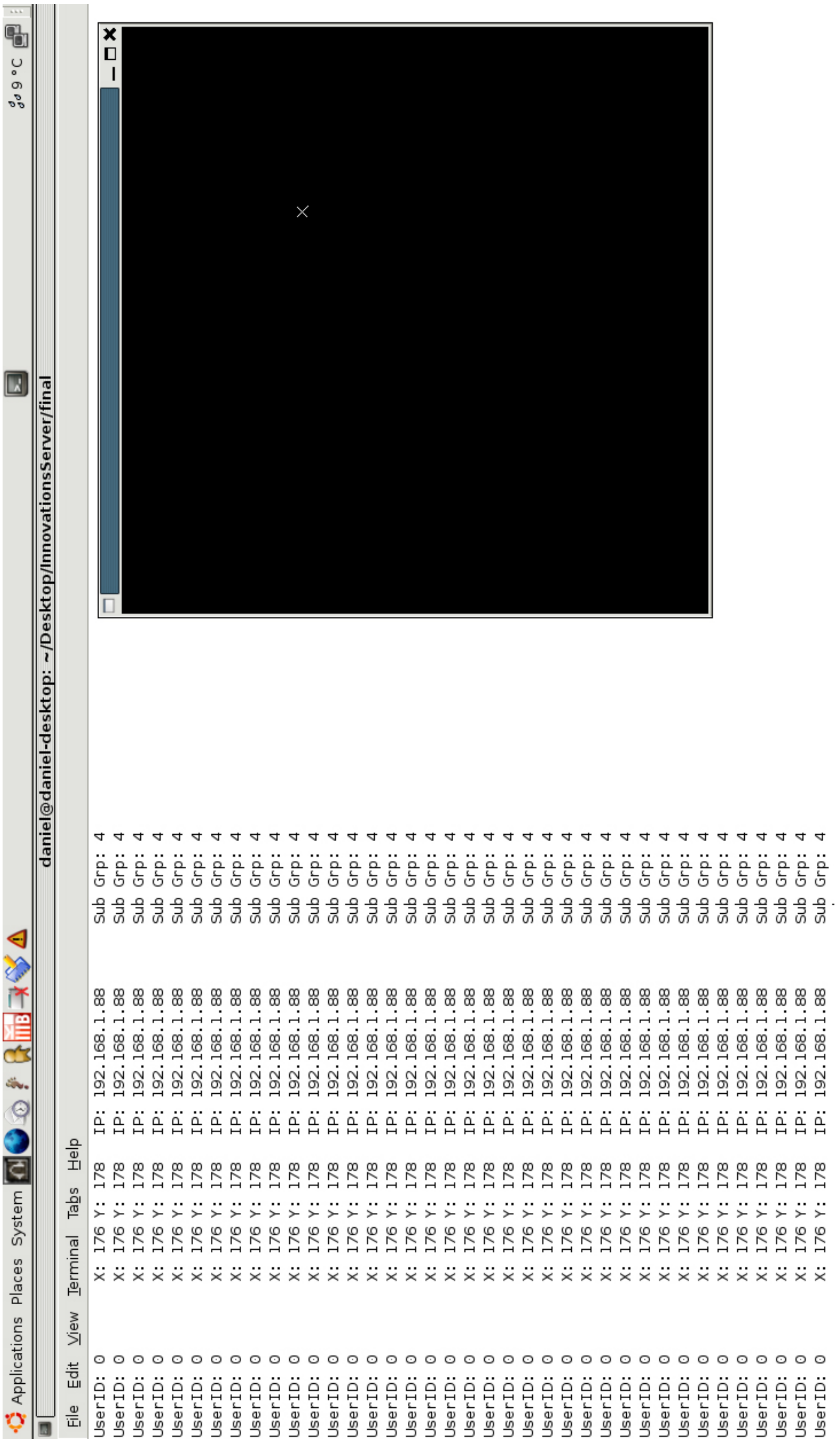

Figure 4.2: The server with one connected client - represented by the white cross.

Bibliography

Amero, J. (2008), 'Introduction to nintendo ds programming'. http://-patater.com/files/projects/manual/manual.html.

Anonymous (2008), 'Osdl - a guide to homebrew development for the nintendo ds'. http://osdl.sourceforge.net/main/documentation/misc/nintendo-DS/-homebrew-guide/HomebrewForDS.html.

Anonymous (undated), 'Homebrew programmers guide to the nintendo ds'. http://www.dspassme.com/programmers_guide/tutorial/index.html.

Assiotis, M. & Tzanov, V. (2005), 'A distributed architecture for massive multiplayer online role-playing games'. pdos.csail.mit.edu/6.824-2005/-reports/assiotis.pdf.

Athens, E. (2008), 'Athens for education'. http://www.athens.ac.uk/.

Bernier, Y. W. (2005), 'Latency compensation methods - in client/server in-game protocol design and optimization'. http://www.resourcecode.de/stuff/clientsideprediction.pdf.

Blizzard (2006), 'World of warcraft'. http://www.worldofwarcraft.com/.

Bogojevic, S. & Kazemzadeh, M. (2003), 'The architecture of massive multiplayer online games'. http://graphics.cs.lth.se/theses/projects/mmogarch/-som.pdf.

Budgen, D. (2003), *Software Design: Second Edition*, Pearson Education Limited, Edinburgh Gate, England.

Caltagirone, S., Keys, M., Schlief, B. & PhD, M. J. W. (2002), 'Architecture for a massively multiplayer online role playing game engine'. http://-www.resourcecode.de/stuff/clientsideprediction.pdf.

Cox, A. (2006), 'Nintento ds development notes and information'. http://-www.cs.ucl.ac.uk/students/Andrew.Cox/.

devkitPro (2008), 'devkitpro'. http://www.devkitpro.org/.

Donahoo, M. J. & Calvert, K. L. (2001), *TCP/IP Sockets in C - Practical Guide For Programmers*, Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205.

Double, C. (undated), 'Homebrew nintendo ds development'. http://-www.double.co.nz/nintendo_ds/.

Eisenhauer, G. & Schwan, K. (2006), 'Publish-subscribe for high performance computing', *IEEE Internet Computing*.

Hull, A. (undated), 'Nds/tutorials'. http://www.dev-scene.com/NDS/-Tutorials.

LiraNuna (undated), 'Liranuna's development blog'. http://-liranuna.drunkencoders.com/nds-2d-tuts/lesson-2/.

neimod@hotmail.com (undated), 'Ds tek - nintendo ds technical information'. http://neimod.com/dstek/.

SAMS (1999), *Teach yourself C++ in 21 Days (3rd Edition)*, Sams Publishing, 201 West 103rd St, Indianapolis, Indiana, 46290 USA.

Schmidt, D. C. (2002), *C. C [plus plus] network programming Vol.1 : Mastering complexity with ACE and patterns*, Addison-Wesley, Ste . - Boston; London.

Smed, J., Kaukoranta, T. & Hakonen, H. (2002), 'A review on networking and multiplayer computer games', *TUCS Technical Report*. http://staff.cs.utu.fi/ jounsmed/papers/TR454.pdf.

Supercard (2006), 'Supercard products'. http://eng.supercard.cn/.

Vijn, J. (2007), 'Tonc: regular tiled backgrounds'. http://www.coranac.com/-tonc/text/regbg.htm#sec-map.

Appendix A

Tank Wars DS - User Manual

A.1 Source Files

The source files, this report and executables for this project can be located on the supplied CD:

Server Executable:

/PC_Server/serverV1.0

Server Source Files:

/PC_Server/source/

Nintendo DS Client Executable:

/DS_Client/twDSClient.nds

Nintendo DS Client ARM9 Source Files:

/DS_Client/arm9/source/

Nintendo DS Client ARM7 Source Files:

/DS_Client/arm7/source/

PC Client Executable:

/PC_Client/twPCClient

This report:

/Documentation/

A.2 Running the PC Server

This should run on any linux based computer, however it has not been tested on any other machine other than the one stated in Section 3.2.

To run the server type ./serverV1.0 *name* - where *name* is the DNS name or IP of the machine the server is to run on (e.g wg0712). This should then run and create a small 512x512 openGL window where you can see connected users represented by crosses. The server should then be in a state to accept clients. Press **F1** to maximize the window, and **ESC** to quit.

A.3 Running the PC Client

To run the PC client type ./twPCClient *name* - where *name* is the DNS name or IP of the computer running the server. Press the arrow keys to move the cross representing the player around the window. Press **ESC** to quit. The server must already be running on the host machine before running the client.

A.4 Running the DS Client

The DS client requires a 3rd Party Slot 1 or 2 device to load the .nds file onto in order to run the code. It is possible to run this executable using emulation applications, but it has not been tested on such software. In order for it to work correctly the server must be running on the hosts machine.