

MAVERIK DOOM — Third Year Project Report

Chris Whitworth
University of Manchester

Alan Murta (Supervisor)
Advanced Interfaces Group, University of Manchester

Spring 2000

Abstract

MAVERIK DOOM is the title of the third year project assigned to Chris Whitworth and Matthew Craven.

This report provides an overview of the **MAVERIK DOOM** third year project, with particular regard to the design, implementation and criticism of the network system (i.e., the parts of the system as written by Chris Whitworth).

Contents

List of figures	iv
1 Overview	1
1.1 MAVERIK	1
1.2 Doom and similar games	2
1.3 The project	3
1.3.1 Motivation	3
1.3.2 Brief overview	3
1.3.3 Division of work	4
2 Planning	5
2.1 Design Philosophy	5
2.2 Working together	5
2.2.1 Naming considerations	6
2.2.2 Code separation and integration	6
2.3 Thinking about networking	7
2.3.1 Models	7
2.3.2 Client-heavy	8
2.3.3 Server-heavy	9
2.3.4 Discussion	10
2.4 Thinking about AI	11
2.4.1 The Think Loop	11
2.4.2 Awareness	12
2.4.3 Decision making	13
2.4.4 Action	14
2.4.5 Where does the AI fit in?	14
2.5 Project Management	14
3 Implementation	16
3.1 Walkthrough	16

3.1.1	Connection and Initialisation	16
3.1.2	Movement	17
3.1.3	Firing	17
3.1.4	Disconnection	17
3.2	Implementation details	18
3.2.1	Network layer	18
3.2.2	Server side	19
3.2.3	Client side	20
3.3	Artificial Intelligence	21
3.4	Testing and bugfixing	22
4	Discussion	23
4.1	Evaluation with respect to project brief	23
4.2	Evaluation with respect to gaming issues	24
4.2.1	Speed issues	24
4.2.2	Causes: Polling	24
4.2.3	Causes: Streams	24
4.3	Other comments	25
5	The future	26
5.1	Speeding up the network code	26
5.1.1	Polling	26
5.1.2	Streams	27
5.1.3	Other areas	27
5.2	Extensibility of the network code	28
5.3	Artificial Intelligence	28
5.3.1	Research	29
6	Conclusions	30
A	mxserver.c source	32
B	mxserver.h source	49
C	networkclient.c source	51
D	networkclient.h source	67
E	networkincludes.h source	69
F	taunt.c source	70

G taunt.h source	72
H commandsplit.c source	73
I commandsplit.h source	75
J configure perl script source	76

List of Figures

1.1	A screenshot from the MAVERIK legible city demo	2
1.2	Screenshot from Unreal Tournament	3
6.1	Screenshot showing two MAVERIK DOOM clients running alongside the server on Chris' machine	31

Chapter 1

Overview

This chapter provides background information on MAVERIK, Doom and similar games, and the project itself. We will also briefly discuss the work involved, and the division of labour between myself and Matt Craven.

1.1 MAVERIK

MAVERIK is the MAnchester Virtual EnviRonment Interface Kernel — that is, it is an API which provides a set of functions to manage objects within a virtual environment. MAVERIK runs on Unix and Microsoft Windows based computers.

MAVERIK itself doesn't actually do any rendering itself; it simply 'manages' the objects. Any object within MAVERIK can (and, in fact, should) implement a render callback, which is used to ask the object to render itself howsoever it sees fit. Normally, MAVERIK will use OpenGL® as the rendering back-end, although there is no reason why this has to be the case.

A large number of utility functions are provided in addition to the object and world management routines — for example, there are many basic 3D geometry functions including vector/plane intersections, transformations, and much more.

Existing software produced using MAVERIK includes a several virtual cities, and a walkthrough of an oil rig. It has also been used along with a network layer known as 'Deva' to create an online virtual multiuser environment.

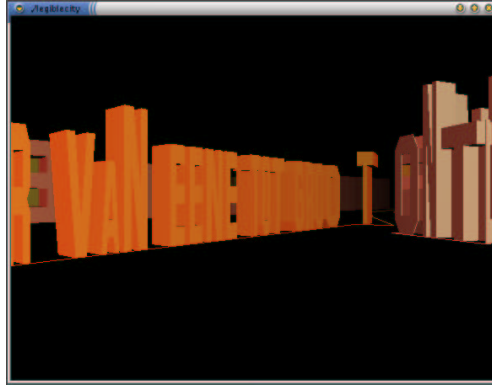


Figure 1.1: A screenshot from the MAVERIK legible city demo

1.2 Doom and similar games

The original Doom was amongst the first of a style of computer games known as the ‘first person shooter’, or FPS. As the name suggests, this genre of game places the player in control of a character who basically has to run around shooting things (usually aliens, barons of Hell or their own workmates!); the action is viewed through the eyes of the player to provide a first person perspective on the action.

Doom itself was a bit of a cheat in that (along with its predecessor Wolfenstein 3D) did not in fact present the player with a real 3D world. Instead, a 2D map with height values for the floor and ceiling was used, and a raycasting algorithm used to generate a ‘3D’ view of this world.

The first (well, best known) ‘true 3D’ FPS is Quake, which, in contrast to Doom, actually used a real 3D world, including full 3D models for the player, enemy characters, collectable items, and so forth.

One massively popular feature of many of these games is the ability to play them across a network with friends, colleagues or even complete strangers as part of a ‘deathmatch’ game. Here, the objective is simply to run around notching up as many kills (‘frags’) as you possibly can. This particular genre recently came to the fore with the release of Quake3:Arena and Unreal Tournament, both of which are multiplayer games only (although both games have advanced artificial intelligence ‘bots’ to enable users to practise when not connected to the network).



Figure 1.2: Screenshot from Unreal Tournament

1.3 The project

The project brief was as follows:

“The Maverik VR system developed by the Department’s Advanced Interfaces Group (AIG) enables the rapid development of interactive 3D graphics applications. It’s about time someone tried to use Maverik in a ‘games’ context, and a game of the Doom / Quake genre is an appropriate candidate.”

1.3.1 Motivation

MAVERIK provides a way of dealing with objects in a 3D environment which lends itself very naturally to games programming. In addition, many of the utility functions provided are also extremely useful within a games programming context. Up until now, however, there had been no test of MAVERIK in a gaming situation. The motivation behind the project was therefore to design and implement a game which could effectively demonstrate MAVERIK’s suitability to managing a gaming environment.

1.3.2 Brief overview

It was decided that a simple multiplayer deathmatch game, along the lines of Unreal Tournament or Quake3:Arena would be implemented; support for play across a network and AI bots would be incorporated. Additional features could be planned and added as the project progressed.

Development was to be undertaken using Unix-based machines (largely Linux machines, although the Advanced Interfaces Group also had several Silicon Graphics machines available for testing too¹).

1.3.3 Division of work

The project was to be undertaken by two people, myself and Matt Craven, under the supervision of Alan Murta of the Advanced Interfaces Group. Given that Matt had a large amount of prior experience of graphics coding, and I had some (albeit highly limited!) experience of network coding, it was decided that the project should be divided as follows:

- Matt Craven: Graphics engine, navigation and player control, weapons.
- Myself: Network code, Artificial Intelligence².

¹However, compiling the project on these machines proved quite a challenge and the full project was never tested on an SGI.

²However, due to unforeseen difficulties with writing the network code, time ran out and the full AI code was not implemented. Matt wrote some rudimentary AI code for testing of multiplayer support before the network code was fully implemented, and some ideas and designs regarding how the AI could work were, however, drawn up.

Chapter 2

Planning

Although much of the development of the project was fairly ‘organic’, given the fairly broad nature of the project overview, along with the fact there were two people working on it, some element of planning and design was obviously needed.

2.1 Design Philosophy

As mentioned, the project was very open ended, and as a result the development tended to follow a heavily prototype-oriented development model. Certain aspects of the project were fixed at the beginning (i.e., the separation of the work, as detailed above, and how the different sections of the project would work together, as detailed below) but for the most part the design evolved as the project progressed. More detailed design decisions — for example, the APIs used for each module in the code — were made as the project progressed.

For example, with regard to the network code, the basic protocol for communication between the server and client was established at the beginning, but was designed in such a way as to leave it as a fairly open standard which could be extended as and when needed (as, in fact, was necessary as development progressed).

2.2 Working together

The fact there were two people working on this project brought several additional issues into play over and above a normal project. They are outlined below.

2.2.1 Naming considerations

Within the code, it is likely that we would want to use some variables with similar or identical names (for example; counter, x, y, etc). To avoid the inevitable confusion this would cause, a naming convention for variable and function names was decided upon.

The code itself was to be split into multiple source files for ease of management (for example, the network code for the game was held in *network-client.c*, the weapons code in *weapons.c*) and each source file had an associated header file (as per normal C language development). It was decided, therefore, that any variables or functions declared within a source file that were to be visible outside of that source file should be prefixed by a unique identifier (usually the name of the file without the extension), followed by an underscore. In several cases, it was decided that variables were not ‘centralised’ enough to warrant tying to a particular file, and in these cases an exception was made and the prefix dropped.

2.2.2 Code separation and integration

As previously mentioned, the code was separated into multiple source files. The code was also designed and written to be very modular — as far as possible, the two halves of the code were to be self-contained, and should simply present function interfaces to be called as necessary. Matt also made extensive use of callback systems to allow, for example, commands to be added to the console system easily without needing to actually alter the console code anywhere.

This made for much easier code maintainance and also much simpler bug tracing — in the event that non-obvious bugs crept into the code, a test bed could easily be written in order to ‘simulate’ the rest of the game. Indeed, this was done on one occasion to track down a mysterious segfault in the network code — a simple test bed which allowed the developer to enter commands explicitly into the client and watch the results was written and in this way the segfault was located and eliminated.

In this way, also, updates to the code could be made without needing to redistribute the entire source of the project — just the changed source files. There were occasions during development when this was not possible—obviously at some point elements of the two areas of the code will overlap—but as far as possible, these eventualities were minimised and the code integration progressed seamlessly.

2.3 Thinking about networking

One of the first tasks faced was that of deciding on a network model for the game. It was decided from the start to use a client-server model — this allows a much more efficient distribution of data across the data, as if we use a direct-connection model (where every client connects to every other client) the number of connections quickly becomes unmanageable (let alone the amount of work needed to work out how you would announce a new connection and get clients to open up connections to them!)

A networked game can be thought of, at the basic level, as a group of player characters interacting within the gameworld. The characters are under the control of either humans or AIs and are pretty much free to do as they like. The gameworld, however, remains pretty much fixed throughout the duration of the game, and imposes a number of rules on how the characters behave.

In the first place, we needed to establish what a networking model for a multiplayer game must achieve. The overriding factor in a game is that it should be playable — that is, it should run at a reasonable, consistent speed (realtime interaction is kind of implied here!), and the gameworld should be consistent across all clients (it could be very confusing if two clients are slightly out of sync, meaning that one client thinks that someone got shot but someone else thinks they didn't!) The networking code should therefore be totally transparent to the players — it should not impede their ability to play the game in any way, and as such should not place unreasonable loads on the processor or the network¹.

2.3.1 Models

There are two extreme models which can be taken as example theoretical network models, the client-heavy model and the server-heavy model. As we shall see, whilst both these models are theoretically sound, real-world limits (particularly things like available network bandwidth and processor time) mean that a compromise somewhere between the two had to be arrived at.

¹This is the theory anyway! We shall see soon that the network model arrived at is lacking in a number of these key areas — there are a number of suggestions as to how it could be improved toward the end of this report.

2.3.2 Client-heavy

Overview

Under this model, each individual client is responsible for maintaining their own copy of the gameworld and ensuring everyone else has an up-to-date copy. The server simply acts as an arbitration engine and a synchronisation system. Historically, this is the kind of model most commonly used in multiplayer gaming.

Implications

- The clients communicate by passing information about the movement of the objects under their control via the server to all the other clients. The server merely passes the data between clients.
- The server can therefore be very simple, and will probably in fact run on one of the client machines (as is the case with most modern games)
- The network bandwidth requirements are (in theory) relatively low for this model - only the movements of characters and non-player objects is broadcast over the network. It is up to the client to decide what to render and so forth.
- The clients will be fairly complicated pieces of code, and will have to be 'trusted' to follow the gameworld rules.
- There is an issue in deciding who would be responsible for controlling non-player objects (such as particles) and deciding when someone has been shot or when any other kind of interaction occurs.
- The most immediate problem with this system is one of determinism and concurrency. As each client is responsible for running their own copy of the gameworld, there is a possibility that clients running on different speed machines may get out of sync. The server would therefore be responsible for ensuring the worlds were synchronised to a central clock.
- There is a possibility that the system also assumes that the gameworld is deterministic (in terms of non-player characters) — if the clients are responsible for updating all objects themselves, then there can be no random element to the game (unless it is seeded beforehand) otherwise everyone would see a slightly (or even significantly!) different version of the gameworld.

- The client has full awareness of the whole gameworld at any point in the game. This means that bots can also be fully aware of the gameworld — which makes writing the bot AI much easier, but could also give the bots an unfair advantage of the human players.
- Maintaining a complete copy of the gameworld for every bot would probably be very expensive in terms of CPU time, and if you want several bots running (which is very likely to be the case) then it could be considered overkill to have each one running its own copy of the world.
- There is therefore the possibility of having another, separate, bot server running as a world-aware client which could then provide an abstraction layer for bots similar to the interface as described in the server-heavy model (see below).

2.3.3 Server-heavy

Overview

In this model, the client has no awareness of the gameworld per se. The server provides the client with a list of objects that are currently visible to the player, and the client returns player interactions with the gameworld back to the server. To put it in relatively crude terms, think X Windows or VNC, but for gaming.

The server is therefore responsible for maintaining the gameworld state in a legal fashion, and the client can (pretty much) consist solely of a rendering engine and a user interaction engine.

Implications

- The clients are (relative to the other model) simple. They will have a minimal processor load on the client machine (assuming the display loop is of minimal processor load — an artificial situation, but for the purposes of this consideration we can assume that it is)
- Conversely, the server will be a huge, monolithic engine and will probably require a very large amount of processor time on the serving machine.
- The server has to handle all interactions between objects in the world — collisions, movements, the tracking of non-client objects (missiles, etc), concurrency checking, etc.

- The network will also be very heavily loaded from the server to the client, as for every point in the game, the server has to send out details of all the objects a player can see — depending on the level of abstraction used, this could be anything from one or two to several thousand.
- The network load from client to server, however, will be low, as the only data being sent back is effectively user interaction data.
- All bot AI has to be totally perception based — that is, based on what the bot can see at any one time in the world.
- This makes objective based bots very hard — they have no concept of the whole game world and therefore this makes it hard for bots to navigate around the game world or have any kind of long-term objectives. A progressive map building algorithm could be used, but this would probably require more effort than was strictly warranted.
- In a way, this means the bots will have to play ‘fair’ — and will therefore probably be very easy to beat!
- In theory, concurrency between clients is not really an issue. In practise this isn’t quite true, as it is constricted by the availability of network bandwidth.
- This is somewhat of an idealised solution — in theory, with flexible enough protocols a generic client could be used with a large number of different games. However, the enormous amounts of network bandwidth required combined with the difficulty of writing good bots means that it is probably highly impractical.

2.3.4 Discussion

Neither of these models are in fact ideal given real world constraints; they both represent extreme views of how one could implement a networking model for a multiplayer game. The actual networking model decided on was a partial compromise between the two (although the client-heavy model is the one which the actual model is primarily based upon).

In an ideal world where processing power, network bandwidth and so forth are unlimited, the server-heavy option would be a really nice idea — the issues of asynchronicity effectively disappear because there is only one

copy of the gameworld in existence, and the client side of things is kept as simple as possible.

In reality, people want to play these types of games on their home PCs over slow networks (usually dialup connections) so this option is utterly impractical. The client-heavy model is therefore the preferable option in this situation.

The eventual (proposed) network model is briefly detailed below. The actual model implemented differs from this slightly because of unexpected issues discovered whilst developing the software:

I would propose for the purposes of this project that we should probably lean towards the smart-client model: we would have a fully ‘world aware’ client that maintained its own integrity checks and so forth. The data transmitted between clients would have to be assumed to be valid (unless a certain amount of intelligence was built into the server².) and non-player objects would be managed by the client that generated them. Non-player generated objects (ie, doors) will have to be managed by each client individually, although the triggers would be broadcast and synchronised globally to ensure each client still had the same view of the game world.

2.4 Thinking about AI

The artificial intelligence routines for a game like this are complex. It would be easy to get bogged down in the detail of the problem if we launch straight into it. Initially, a more abstract approach is needed: It is much easier to think about if we break the AI down into a set of conceptual requirements, which we will call a ‘think loop’:

2.4.1 The Think Loop

- The bots need to have some concept of ‘awareness’ of the world — i.e., they need to be supplied information about the world with which they are interacting by the main game engine.

²We could, for example, also have the server retaining a copy of the gameworld and performing its own integrity checks. This would lessen the risk of ‘rogue clients’ crashing the game by sending invalid data, or causing the worlds to become out of synch.

- The bots also need some concept of their current situation within the world — their current state in terms of position, view direction, and so forth.
- The bots possible also need some concept of previous situations, actions taken and consequences of these actions.
- The bots then need to make decisions based upon the information supplied, their current state and in light of previous knowledge acquired.
- These decisions are then translated into actions and submitted back to the game engine for validity and integrity checking. The think loop then restarts.

2.4.2 Awareness

MAVERIK provides a wonderfully useful function for this kind of thing: `mav_SMSCallBackExecFnExec()`. This function takes a set of clip planes (a function is also provided to generate these from view frustums), an SMS³ and a function callback. It then clips the objects within the SMS to within the clip planes, and calls the callback function provided on the objects that are within the clips.

If we simply use this callback function to add each object to a new object list of currently visible objects, we can build up a simple ‘vision’ system.

MAVERIK also allows customisable object types, so we could define different object types for, e.g., players, walls, collectable objects and so forth. This would allow for some kind of symbolic conceptual visual system, rather than a solely visual one. This is useful, because the human visual processing system is symbolic, rather than subsymbolic — that is, we see a red cup on an oak table, rather than a red hollow cylinder at with a half torus of next to it, on a large flat textured box, with four tall thin boxes at each corner underneath it. The level of detail in a subsymbolic system is unnecessary and, in fact, would probably be less useful in this kind of AI system.

The current state of the bot would be dealt with by the navigation/player management system (assuming that the bots would be implemented as standard players — the player management system as implemented by Matt allows a player to implement a ‘think’ callback; in the case of a human player, this would simply be a navigation system) and would be accessible as a set of structures from the bots ‘think’ function.

³Spatial Management System - think of it as a set of objects.

The concept of a history is slightly harder to implement. At one level, the bot could attempt to build up a map of the level whilst navigating around it by interpreting and storing the locations of objects of the ‘wall’ class in the world. Over and above this, one could start using neural networks and self-organising feature maps to cluster information about current world state and previous actions taken. The exact information to be stored would be extremely dependent on how the thinking was actually performed (e.g., in a state-based AI we could record the result of certain state transitions and make decisions about whether the outcomes are considered good or bad in light of the consequences for the bot). One possibility is that the bot could store a history of where it sees other players clustering, or where it last saw a particular player.

2.4.3 Decision making

This is where the complicated stuff starts! There are any number of algorithms we could use here to control the actual behaviour of the bots. We shall attempt to suggest a few that could be considered for implementation in a game such as this.

About the simplest algorithm we could implement is a two-state bot with no concept of history whatsoever. The two states are ‘seeking’ and ‘chasing’. In the seeking state, the bot moves in a pseudo random fashion (some probability distribution code could be used to ensure that it didn’t cluster around a particular area), until it sights an enemy player. It then enters the ‘chasing’ state, where it follows the other player, firing at them.

This is a very simplistic algorithm. It falls down in that there is no survival instinct in the player. We can extend it therefore to incorporate some kind of retaliation/avoidance code: for example, in the event of the bot being shot, it will compare the power of the weapon it has been shot by with the weapons it is carrying. If he has more powerful weapons, the bot can take the gamble, seek the player who shot him and attack him. If he has less powerful weapons or is very low on energy, the bot could attempt to run away and hide!

A more advanced bot could build up a map of the world progressively, and include concepts such as a memory of where the other players tend to cluster. Routefinding could be implemented so that a bot knows how to get to another player on the other side of the world — this would be made easier by the inclusion of ‘waypoint’ objects; that is, non-renderable objects that the bots can navigate in between.

2.4.4 Action

The decisions then made can be easily translated into actions and sent to the navigation system. This system has been implemented by Matt in such a way that you submit your desired action to the navigation function, which then checks to see if it is valid (i.e., it performs collision detection and so forth), alters the movement as necessary to allow it to be valid, and updates the player state as necessary.

Therefore, the bot can simply issue an ‘I’d like to run forward now’ command, and the navigation system will deal with the rest. Obviously, however, this will mean that not all of the requests will be dealt with as expected. Any state-based AI routines would therefore have to take this into account and not assume that previous actions were necessarily carried out totally as expected.

2.4.5 Where does the AI fit in?

We also need to think about at level the AI code will fit in with the game code — there are a number of possibilities here.

- The bots are standalone clients which connect to the server and, for all intents and purposes, look exactly the same as any other client.
- The bots are integrated into the clients, effectively piggy-backing on a players connection, using the same functions and so forth.
- An abstraction layer could be used — that is, a single client is used, but many bots could be spawned from it.

From a design point of view, the first option is the nicest — it keeps the code nice and separate. It does, however, build in a certain level of redundancy — this is where the second and third options come in.

2.5 Project Management

The entire project is written in C, with building of the code controlled by a standard GNU Makefile. However, because the installation of MAVERIK is implemented in such a way that the libraries and header files do not have a fixed location, it was deemed necessary that some form of makefile auto-generation was necessary. The autoconf, automake and associated utilities were considered, but it was decided that this was probably overkill for this

relatively small project (it would have required the writing of many extra macros to locate the MAVERIK libraries).

In the end, a short perl script was written to auto-generate the Makefile. The script searches existing library paths to see if it can determine the location of MAVERIK in this way; if it fails, it asks the user for the location of their copy of MAVERIK. Having done this, it substitutes the location of the libraries and headers into the skeleton Makefile (included as a here document in the script) and writes it out.

Chapter 3

Implementation

This chapter focusses on the details of the implementation of Chris' half of the **MAVERIK DOOM** project — the design of the network code protocols, any differences to the original designs, and other issues faced during development.

3.1 Walkthrough

In order to get an overview of the networking system, the easiest thing to do is probably to give a high-level walkthrough of what would happen in a typical multiplayer session.

We begin by starting the server process on an arbitrary machine. The server initialises a number of global states; it also holds a copy of the player status array, which is initialised to empty. It also maintains an array of currently connected clients, and the associated global IDs of the players.

3.1.1 Connection and Initialisation

A client is then started, and a game begun. The network state of the client is set to 'NotConnected'. The client can then connect to the server, and issue a login request giving details of the desired player name, the current location, view direction and animation frame of the player; the client enters the 'LoggingIn' network state. The server adds these details to its player status array, generates a global ID for the player and sends it back. Upon receipt of the global ID, the client enters the 'Connected' state.

The server then steps through the array of currently connected clients, and issues a new player message to all except the new player. It also an-

nounces to the new player the state of all the currently connected clients (it does this by issuing ‘new player connect’ commands to the new client — the process is exactly the same as far as the client is concerned; the client doesn’t need to know that they were there all the time).

3.1.2 Movement

The player then moves around within the gameworld. As soon as the player has moved by an amount greater than a certain threshold value, a move command is issued to the server giving the details of the new player location, view direction and animation frame.

Upon receipt of this command, the server updates its local copy of the player states, and then steps through the client array, sending the new state out to all connected clients.

3.1.3 Firing

When a player fires, a message is issued to the server detailing the origination point, direction and weapon type of the shot. The server sends this out across the network. The other clients then create a ‘dummy’ projectile which does no damage and causes no recoil effects upon impact, but which in all other respects is treated exactly like a normal projectile.

If the bullet is considered to have hit something by the issuing client, then a hit message is issued to the server. The server then distributes this to the other clients. The appropriate client (ie, the one that has been hit) then calls an appropriate function to do damage to themselves, move the player in accordance to recoil effects, and so forth. Blood and explosion effects are displayed by all the clients.

If the hit causes a players energy to fall below zero — i.e., the player is killed — then the player in question issues a command to the server to say that they have been killed by the firing player. The server can then update the scores appropriately, and a message is then sent to all the connected clients (“Player1 was killed by player2’s rocket launcher” or similar).

Periodically during the game, the server will send a message to all players announcing who current winning player is and what their score is.

3.1.4 Disconnection

If the client quits, dies, or explicitly requests to disconnect, a message is sent to the server announcing this. The server removes the player from the player

array, closes the connection and announces the disconnect to all the other clients, who then in turn remove the player from their player array.

3.2 Implementation details

3.2.1 Network layer

The network code was implemented using TCP/IP networking, using the standard stream based POSIX network code. This was for reasons of simplicity more than anything else — although the author had no direct experience of networking code before this project, a brief survey of opinions suggested that it would be much simpler to implement it in this way than in any other, and shouldn't make too much difference in speed¹.

The commands are implemented using simple text strings, of the following format:

command:param1:param2:....:paramN

That is, the command begins with the command name, followed by a list of parameters separated by colons. A simple parse routine was written to split a string on a given character and return an array of strings².

By and large these commands are the same whichever way they are being sent, although the client requires the command to be in upper case, and the server requires it to be in lower case. This was done so that during development, the network activity could be streamed to a file (or to stdout) and monitored easily, and the direction of the commands could easily be determined.

Briefly, the commands are:

¹This actually turned out to be untrue — or at least, the decision to use stream-based networking was probably, in retrospect, a bad one. This is discussed in more detail in the Discussion chapter

²Ok, char*s

Cmd	Issued by	Details
hlo	Client	Announces a client to the server
con	Server	Announces a connection.
id	Client	Used by server to tell the client its unique global ID
say	Both	Say something to the world
mv	Both	Movement
fr	Both	A shot has been fired
bye	Client	Disconnects from the server
dis	Server	Announces a disconnection
hit	Both	Someone's been shot!
msg	Both	A generic message format. Used for anything else.

3.2.2 Server side

The decision was taken to effectively make the server side of the system become a multi-command message exchange system — at the simple level, the clients would issue a command to the server about what had changed or happened in their gameworld, and the server would then issue this command to all the other clients, who would update their world accordingly.

In actual fact, the server required a certain amount of intelligence itself in order to arbitrate who to distribute the information to, keep track of scores, who is connected and so forth. As a result, some concept of global object IDs was required.

Each client keeps track of the players in the gameworld using an array of `PLAYER` struct types. Because of the way this array is constructed, the index of the same player in the array on different clients is different; therefore, each player is also allocated a global ID by the server at login time.

The server effectively operates in two halves. The first half receives a command, parses it and updates the appropriate local state. The second half then sends the relevant commands out to all the clients. The receiving half simply parses the commands and deals with them (`dealWithCommand()`). This is currently done using a huge block of `if(cond) {...}` statements. It would obviously be better done using a callback system, but the initial concern was to get something working rather than necessarily implement it nicely³. The other half contains a certain amount of intelligence. A number of `networkEvent` types are known, each with their own parameter sets. The `an-`

³It would be relatively trivial to turn it into a callback system.

nounce() function deals with this half of things, using a switch statement to construct an appropriate message for the given event type which is then sent to all the connected clients.

The server also contains a realtime clock running off a signal. This ticks fifty times a second, and sends out a synchronisation pulse once a second (i.e., once every 50 ticks). This is not currently used by the clients, although the theory is that it could be used for adaptive movement thresholding and better synchronisation⁴. The implementation of this timer was in fact one of the more problematic areas of the server code and warrants a little more explanation.

It is implemented using the POSIX signal handling system. It is possible to request that a POSIX compliant OS sends you an alarm signal a given amount of time in the future. You can then attach a signal handler function to `SIG_ALARM`, and this function will therefore be called asynchronously at some point in the future (the granularity of the timer is specified in the POSIX documents). If the function, along with whatever else it does, sets up another timer for the same amount of time in the future, you can call this function at a defined interval in time irrespective of what the rest of the code is doing.

So, our code sets an alarm for every 0.02 seconds, and increments a global clock. Every 1 second, a tick message is sent out.

The problem arises in the use of this along with network code (or indeed any file-oriented code) which is monitoring several connections at once. The **select()** function which is used to monitor several connections for activity will also return on *any* signal — this includes our alarm signal — with an error code. We therefore need a global status flag (in this case *sigAlrm-Caught*) which is set in the function called on `SIG_ALARM`. This flag is checked upon **select()** returning with an error, and if it is set, we ignore the error and re-enter the **select()** loop. If it is not set, we can deal with the error as appropriate.

This sounds relatively obvious in retrospect, but the Linux man pages are staggeringly cryptic about the behaviour of **select()** with respect to signals, and it took a lot of thinking to work out what was going on.

3.2.3 Client side

The client side pretty much mirrors the server side — ie, it sends commands to and receives commands from the server.

⁴See the Discussion chapter for further details.

It implements three console function callbacks - open connection, close connection and say something. The open connection and close connection functions are fairly self explanatory — as well as the network code for implementing these functions, they also issue the relevant commands to the server. A non-blocking connection is used, as the network code is polled — we don't want the entire game to freeze if there's no network activity!

Opening a connection also sets the client state to `WAITING_ON_ID`. In this state, any IDs announced are assumed to be our own, and we then leave the state. Any IDs received outside of this state are ignored.

The main program loop calls the `networkclient_pollNetwork()` function once every game cycle. This function reads any data from the network into a buffer, and then parses the commands received as necessary. The command parsing in this case is done using a callback system.

If a *connect* command is received, the current list of connected players the client has is checked, and if the global ID already exists it is assumed that a previous player is reconnecting, otherwise a new player is created. The status of that player is then copied into the array, and commands regarding that players state will be dealt with in the future.

If a *say* or *message* command is received, the associated message is simply displayed on the console. For a *move* command, the appropriate player position and view is set (the view has to be independently recalculated so the player model can be rotated by the graphics engine), assuming the player exists and isn't the local player.

The *fire* callback determines the origin, direction and weapon type of the projectile in question, and then calls the `weapon_projectile_create()` function to create a dummy projectile as appropriate. This projectile is then just handled by the weapons system as would a normal local projectile, except all non-graphical effects of collisions are ignore.

Upon receiving a *hit* command, the client checks to establish if the player hit was itself. If it was, the `weapons_player_hurt()` function is called with the details of who fired, who was hit, in which direction the projectile was travelling and what type of weapon it was that fired the projectile.

3.3 Artificial Intelligence

As mentioned in the introduction, the AI was never implemented due to lack of time. Rudimentary bot code was added by Matt, and the model used allows the use of AI players relatively simply. However, the way the existing bots were implemented effectively means they cannot be used over

the network.

Work began on forking off the bot code from the main client code and running it as a separate client, but this was never completed, again due to time constraints. It should be relatively trivial to complete, however.

3.4 Testing and bugfixing

Because the development cycle was heavily prototype oriented, and because there was working code to play with almost from day one, testing occurred during the entire development cycle.

The decision to use stream-based TCP/IP networking meant that the server could be interrogated via a telnet interface, which was invaluable during the development of the server. The server is also able to output a large amount of debug information to a log file if run in verbose mode (it is reported to stdout if the `-v` switch is used). The client also had a large amount of debugging trace code in it during development, although this got stripped out towards the end in an attempt to streamline the code.

Testing of the network code was possible even on one machine because it is quite possible — indeed, it is usual — to run MAVERIK applications in a window. Several copies of the client code could be run at once, along with a copy of the server, on the same machine. Testing was also done using several machines on the same network at once, and as many permutations of location of clients and servers relative to each other were tried as were practical.

During development, it was often necessary to use the standard array of Unix debugging tools to trace faults and crashes in the program. As mentioned in the introduction, at one point it was necessary to write a dummy test-bed for the network client code to trace a series of segmentation faults in the program.

Chapter 4

Discussion

In this chapter we shall evaluate the project in the light of its aims and examine areas of possible improvement.

4.1 Evaluation with respect to project brief

At a fundamental level, the project at least works — that is, the code compiles, runs, and does pretty much what it claims to. The players can run around the world and interact over the network. Technically, it all works.

And it certainly shows that MAVERIK can be used in a gaming context — none of the problems detailed in the next section are down to MAVERIK; they are all areas which, with the benefit of hindsight and given sufficient time, could easily be corrected.

However, it should be noted that the network code made use of almost none of the MAVERIK API — indeed, the server code didn't even include the header files or link against it in any way. Thus it is questionable as to whether my half of the code could in fact be evaluated in this way anyway.

Also, many of the graphic effects implemented by Matt (for example, the sprite system and the player model renderer) were written directly using OPENGL® rather than using a MAVERIK wrapper. There is no real theoretical reason why a new MAVERIK object class could not have been written for each of these, but it is possible that the extra overhead introduced by doing this would have caused yet more performance issues.

4.2 Evaluation with respect to gaming issues

However, one of the major design issues in coming up with the network model was that the game should ultimately be playable — and it is here that things fall down. Whilst the design of the network model is theoretically sound, there are a number of areas in the implementation is lacking.

4.2.1 Speed issues

Most notable of all these areas is that of speed. The game is simply not playable in a real context because of the massive lag that is noticed when playing across the network — it is not unusual for there to be a lag of up to 30seconds to build up between clients.

This obviously leads to serious issues of lack of concurrency between the clients, and bizarre effects such as people getting hit and killed by players a minute after they were actually in the location where they would have been hit can be observed.

The reason for this situation can be traced back to several points in the code.

4.2.2 Causes: Polling

The major reason for the lag is due to the way the polling loop was written.

For simplicity, it was written such that that only one command is parsed per game cycle. As the game cycle executes once per frame, this means that if there is any discrepancy between the framerates of the two clients, the slower client is going to build up a whole stream of unparsed commands in its input buffer. These commands are then only dealt with one per frame.

Also, if there are more than two players connected, the lag gets even worse — even if all the clients are running at the same frame rate, they will receive two commands for every one they parse, and they simply cannot keep up.

4.2.3 Causes: Streams

The decision to use stream-based communication was sound from a debugging point of view; however, from a speed and efficiency point of view it was something of a disaster. As the input buffer is a FIFO pipe — that is, it is accessed in a serial manner from the beginning of the buffer — all data received must be read and processed serially.

The fact that human-readable commands were used also seemed like a good idea at development time — but again, this led to a performance degradation.

4.3 Other comments

There are a number of areas in the code where a particular way of implementing something has been used because it works, rather than because it is necessarily a good idea. As a result the code is rather messy in places — for example, the command parsing code in the server really needs to be a callback system rather than a series of if statements.

I think there may also be a remaining segmentation fault in the code. However, as the error is not repeatable on a predictable basis, and often does not occur at all, tracing it is very, very difficult indeed.

Chapter 5

The future

In this chapter, we will discuss specific points which could be addressed given more time.

5.1 Speeding up the network code

This is the principle problem area in the code. As I mentioned before, the fundamental model behind the network code is reasonably sound — it is because of the way it was implemented that it was slow.

There are a number of ways in which this issue can be addressed. The obvious thing to do first is to address the faults directly mentioned above. Beyond that, there are a number of further techniques that could be used to squeeze further performance out of the code.

5.1.1 Polling

The obvious thing to do here is to adjust the polling code to read more than one command per game cycle. This would be very simple to do — instead of a simple piece of sequential code, the polling function could be replaced with a while loop conditioned on there being data still in the network input buffer.

The alternative would be to run the display routine off an interrupt. This, however, makes assumptions about the speed of the rendering code (i.e., that it is negligible compared to the rest of the code) — whilst these assumptions are fairly reasonable with a good 3D accelerator, in the absence of this, the code would perform very poorly.

Another alternative would be to run all the other routines off interrupts

— that is, just have the main game loop displaying the current gameworld, and then everything else runs in the background. This approach has the added advantage that we don't have to, for example, adjust the speed of projectiles and so forth to account for the current framerate. It is also noticeable that in the game as it stands, the speed with which you move is directly related to the framerate. Again, this problem would be avoided if the movement system could also be run off interrupts rather than as part of the display loop.

5.1.2 Streams

There is less can be done to avoid the problems inherent in using stream based communications. Moving from human-readable commands to binary streams of data would gain some performance improvement. Some simple compression algorithms may also give some speedup.

However, I do not believe that the problems caused by stream-based communication are that significant in comparison to those caused by the polling loop problems.

5.1.3 Other areas

Currently the movement system uses a static threshold value to decide when to announce a player movement to the other players. One way to improve performance would be to use an adaptive threshold — on a busy network with variable network traffic, the load on the network could cause a noticeable slowdown in the game — and, indeed, the traffic generated by a busy game would also cause a noticeable slowdown. Therefore, if we can adapt the threshold to account for this, we can maintain better concurrency across the network (this is not so much a speed improvement; it is more a concurrency maintainance improvement).

The server already has support in it for a timer. If support for this were also added to the client, then this could form the basis for the adaptive thresholding system as follows. The client synchronises with the server. Some time later, it 'pings' the server with what it believes is the current time. From this, the server can establish how long it takes for a message to get to it, and sends this value back to the client, along with what the server believes the current time to be. The client can then work out what the server to client lag is.

From these figures, and knowing the framerate and distance moved per frame, the client can work out how far the player can move in the time it

takes for the position update to get to the other clients. The movement threshold can then be set to this amount. In theory, this would be the optimum threshold value.

Synchronisation could be achieved by halting the entire game and just waiting for several sync pulse messages from the server (ignoring all other traffic). The client clock can then be set to the latest pulse, which should be a reasonably accurate figure for the current worldtime.

The clients could also be made to detect when they are falling out of sync — the server also sends out the time of all movement events, so if these fall out of sync with what the client believes the time to be, the client can resync according to the method described above.

5.2 Extensibility of the network code

Some work could be done on making the network code more modular (i.e., by replacing all the big blocks of *if(cond) {...}* statements with callback systems and so forth) so that as new features get added to the game it is simpler to allow them to run over the network.

For example, one proposed feature is collectable items — the network code as it stands is not easily extensible to allow this, and would require more hackery and kludging to get it to work.

5.3 Artificial Intelligence

Obviously, the most pressing issue is to actually get some written! The authors personal feeling is that the current method of incorporating the bots into the client code is probably the wrong way of going about this — it leads to a large amount of complicated code and requires each client to keep track of exactly who it owns and should be running. It would also mean some restructuring of the server code.

Separating the bot code (as had already begun) into a separate client should not be too hard. Indeed, there is no reason why bots could not be launched from the console, even if they were run from a separate program (launching a bot would simply **fork()** a new process and run the bot with the appropriate server parameters preset).

5.3.1 Research

I think it would be beneficial to do some research in this area before implementing the AI code for real — there are many, many bots for games such as Quake and Half Life available on the internet, and as such, there is a large amount of information available for download on the subject too. An examination of existing techniques would prove valuable, therefore, and possibly avoid wasted effort in trying techniques that have been previously shown to be ineffective.

Chapter 6

Conclusions

What conclusions can we draw in the light of the experience gained from this project?

Well, firstly, MAVERIK provides a theoretically sound basis for writing games — as previously mentioned, there is no reason why, with some adjustments, this project could not become a perfectly playable game.

However, whether MAVERIK could be used for *serious* commercial game releases is doubtful — many modern games rely on programming at a fairly low level in order to gain maximum performance out of a system. Having the overhead of something of the size of MAVERIK in the way would cause major problems.

There is plenty of scope left for developing **MAVERIK DOOM** further too — both in the area of the network code and other areas (for further details, see Matt Cravens report on his half of the project). In a project as open-ended and extensible as this, it is hard to say whether something is complete or not. The only real limits are time and imagination.

That said, the project also stands for itself as a demonstration of what can be achieved using MAVERIK and in that fulfills the project brief — “...It’s about time someone used MAVERIK in a games context...”!

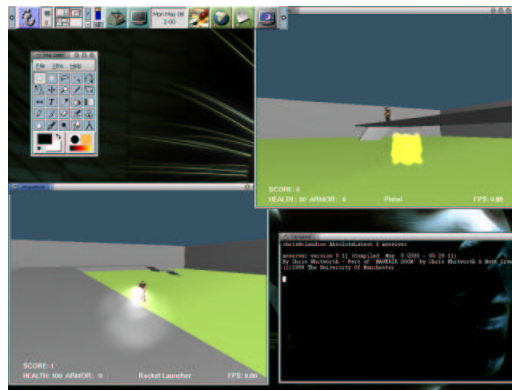


Figure 6.1: Screenshot showing two **MAVERIK DOOM** clients running alongside the server on Chris' machine

Appendix A

mxserver.c source

```
/* mxserver */
/* A simple message exchange server to form the basis of the MavDoom
   networking system */

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

#include "mxserver.h"
#include "player.h"
#include "commandsplit.h"
#include "types.h"

/*****
/* Global variables and stuff */

/* Players table - for convenience sake, we'll keep a local copy so that
   we don't have to ask each client every time someone new connects. Builds
   a certain amount of redundancy in, but what the hey */

PLAYER *player;
int num_players;
```

```

/* Connections table */
Connection connection[MAXCON];

/* A couple of defaults if not set at the command line - log file
   and port */
FILE* LOG = NULL;
int port = 4040;

/* Timer values for synchronisation */
struct timeval serverTicktv;
struct itimerval serverTick;
long int serverTime=0;

bool sigAlrmCaught=false; /* This is really nasty, and I wish there was
   another way to do it. I don't reckon there
   is */

Message announcement;
ProjectileDetails projectile;

char msg_buffer[8*1024]; /* If we ever need to send a command larger than
   8k something is *wrong* */

long int global_object_id_counter=1; /* No object of ID 0 */

/*****
/* Shutdown the server */
void shutdn() {
    int con;
    fprintf(LOG,"Server closing down...\n");

    /* Close all the incoming connections. Not *strictly* necessary as
       the process exiting closes all filedescriptors anyway, but it's
       neater this way */
    for(con=0; con<MAXCON; con++)
    {
        if (connection[con].state!=NotConnected)
        close(connection[con].fd);
    }
}

```

```
}
```

```
/* *****  
/* This will open a socket on port 'port' and start listening for things */  
int startServer()  
{  
    /* Variables... */  
    int serverSocketFD;  
    int serverLen;  
    struct sockaddr_in serverAddress;  
  
    fprintf(LOG,"Server starting on port %i\n",port);  
  
    /* Get us a socket */  
    serverSocketFD = socket(AF_INET, SOCK_STREAM, 0);  
  
    /* Give it a port, set it as an internet socket, and get it listening  
       for connections from anywhere */  
  
    serverAddress.sin_family = AF_INET;  
    serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);  
    serverAddress.sin_port = htons(port);  
    serverLen = sizeof(serverAddress);  
  
    /* And bind the socket to the port */  
    if (bind(serverSocketFD, (struct sockaddr *)&serverAddress, serverLen)!=0)  
        { perror("bind"); exit(1); }  
  
    /* Listen on that socket */  
    if (listen(serverSocketFD, 5) !=0)  
        { perror("listen"); exit(1); }  
  
    /* Nonblocking */  
    if (fcntl(serverSocketFD, F_SETFL, O_NONBLOCK)==-1)  
        { perror("fcntl"); exit(1); }  
  
    /* And update our connection table to take account of the server */  
    connection[0].state = Server;  
    connection[0].fd = serverSocketFD;
```

```

    /* This is a handy thing to know, even though we can look it up in
       the table if we want to... */
    return serverSocketFD;
}

/*****
/* Start the timer */
void startTimer() {

    /* Set the resolution of the timer */
    serverTick.it_interval.tv_sec=0;
    serverTick.it_interval.tv_usec=0;
    serverTick.it_value.tv_sec=0;
    serverTick.it_value.tv_usec=20000;

    /* And get it to send us a SIGALRM when it expires */
    setitimer(ITIMER_REAL,
              &serverTick,
              &serverTick);
}

/*****
/* Signal handler - basically shutdown on anything */
void signalHandler(int signal) {
    fprintf(LOG,"Bailing out on signal %i\n",signal);
    shutdn();
    exit(0);
}

/*****
/* Announce an event to all the connected clients. This might be nicer */
/* if I could make a ServerEvent an object and do this in C++ but a big */
/* switch() { case:; } structure will have to do for now */

/* The exact format of these messages is subject to change. At the moment
   they're human readable. This will probably change :) */

void announce(ServerEvent event, void* data)
{

```

```

/* A few variables */
int connectn,con;
Message* info;
ProjectileDetails *proj;
PLAYER* ply;

/* Prepare the outgoing message */

switch(event) {
case connectEvent: /* A new incoming connection */
    con = (int) data;
    ply = &(player[num_players-1]);
    sprintf(msg_buffer,
"CON:%li:%x:%s:%d:%f:%f:%f:%f:%f:%f:%d:%d:%d\x00",
serverTime,connection[con].hostIP,connection[con].name,
ply->global_id,
ply->pos.x, ply->pos.y, ply->pos.z,
ply->view.x, ply->view.y, ply->view.z,
ply->anim_frame,ply->model_id, ply->weapon_model_id
);
    fprintf(LOG,
"Announcing: New connection by %s from %x\n",
connection[con].name, connection[con].hostIP);
    break;

case disconnectEvent:
    sprintf(msg_buffer,"DIS:%d\x00",(int)data);
    break;

case moveEvent:
    ply = &(player[(int)data]);
    sprintf(msg_buffer,"MV:%d:%f:%f:%f:%f:%f:%f:%d\x00",
ply->global_id,
ply->pos.x, ply->pos.y, ply->pos.z,
ply->view.x, ply->view.y, ply->view.z,
ply->anim_frame);
    break;

case fireEvent:
    proj = (ProjectileDetails*) data;

```

```
    sprintf(msg_buffer, "FR:%d:%f:%f:%f:%f:%f:%f:%f:%d\x00",
proj->plyr_id,
proj->sx, proj->sy, proj->sz,
proj->dx, proj->dy, proj->dz,
proj->speed, proj->weapon);
    break;

case hitEvent:
    proj = (ProjectileDetails*) data;
    sprintf(msg_buffer, "HIT:%d:%d:%d:%f:%f:%f\x00",
proj->plyr_id, proj->hit_plyr_id,
proj->weapon,
proj->dx, proj->dy, proj->dz);
    break;

case timeEvent: /* Announcing a clock tick. */
    sprintf(msg_buffer, "CLK:%li\x00", serverTime);
    break;

case sayEvent: /* Someone said something ("LinuxBoy: Die DozeFreak Scum!") */
    info = (Message*) data;
    sprintf(msg_buffer, "SAY:%li:%s:%s\x00", serverTime,
info->connection->name,
info->message);
    fprintf(LOG, "%s said %s\n",
info->connection->name,
info->message);
    break;

case scoreEvent:
    sprintf(msg_buffer, "SC:%i\x00", (int) data);
    break;

case genericEvent: /* A generic announcement */
    sprintf(msg_buffer, "MSG:%s\x00",
(char*) data);
    break;
}

/* Step through the connection table, and write the message to each
```

```

        open connection */
for(connectn=0; connectn<MAXCON; connectn++)
    if (connection[connectn].state == Talking) {
        write(connection[connectn].fd,
msg_buffer,
strlen(msg_buffer)+1);
    }

}

/*****
/* Signal handler for SIGALRM - reset timer + send event */
void timerEvent(int signal) {
    int k, maxscore=0, maxplyr=-1;
    serverTime++;
    if (serverTime % 50 ==0)
        { announce(timeEvent,NULL); }
    if (serverTime % 500 ==0)
        { char temp[1024];
          for(k=0; k<num_players; k++)
if (player[k].score > maxscore && player[k].active)
    { maxscore=player[k].score; maxplyr = k; }
            if (maxplyr===-1)
{ sprintf(temp,"No-one leads the match"); }
            else
{ sprintf(temp,"%s leads the match with %i frags",
                    player[maxplyr].name, player[maxplyr].score); }
            fprintf(LOG,"%s",temp);
            announce(genericEvent,(void*) temp);
        }
    startTimer();

    /* We have to do this. Explained at the select() in mainLoop()*/
    sigAlrmCaught=true;
}

/*****
/* Remove a connection */

void removeConnection(int con)

```

```

{
    connection[con].state = NotConnected;
}

/*****
/* Generate a global object ID and send it to a connection          */

void generateId(int con)
{
    sprintf(msg_buffer,"ID:%li\x00",global_object_id_counter++);
    write(connection[con].fd,msg_buffer,strlen(msg_buffer)+1);
}

/*****
/* Dump the current status of all the players to a connection      */
/* Done by forcing a whole bunch of players to virtually connect  */

void dumpStatus(int con, int my_pn)
{
    int pn; PLAYER* ply;
    for(pn=0; pn<num_players-1; pn++) {
        if (pn!=my_pn && player[pn].active!=0 ) {
            ply = &(player[pn]);
            sprintf(msg_buffer,
                "CON:%li:%x:%s:%d:%f:%f:%f:%f:%f:%f:%d:%d:%d\x00",
                serverTime,ply->host_id,ply->name,
                ply->global_id,
                ply->pos.x, ply->pos.y, ply->pos.z,
                ply->view.x, ply->view.y, ply->view.z,
                ply->anim_frame,ply->model_id, ply->weapon_model_id
            );
            write(connection[con].fd,msg_buffer,strlen(msg_buffer)+1); }
    }
}

/*****
/* Deal with whatever command we receive from a client          */

/* Rewritten to use commandsplit_Split which makes things far easier :) */
/* This is one horrible big kludge but it works, and that's the main

```

thing. It would be much nicer to do using callbacks but for now I just want the damn thing to work. Which is the main thing. */

```

void dealWithCommand(int con)

{
    char* command = connection[con].buffer;
    char** param;
    int params,k,ply;
    fprintf(LOG,"Command received from %x \"%s\"\n",
            connection[con].hostIP,command);
    param = commandsplit_Split(command, ':', &params);

    /* hlo:NAME:x:y:z:vx:vy:vz:anim */
    if (!strncmp(param[0],"hlo",3))
    {
        if (params!=11)
            { fprintf(LOG,"Bad hlo: command received...\n"); } else {
num_players++;
if ((player = realloc(player, sizeof(PLAYER)*num_players))==NULL)
    { printf("dealWithCommand(): out of memory when adding player\n");
    exit(1); }
player[num_players-1].pos.x = atof(param[2]);
player[num_players-1].pos.y = atof(param[3]);
player[num_players-1].pos.z = atof(param[4]);
player[num_players-1].view.x = atof(param[5]);
player[num_players-1].view.y = atof(param[6]);
player[num_players-1].view.z = atof(param[7]);
player[num_players-1].host_id = connection[con].hostIP;
player[num_players-1].model_id = atoi(param[8]);
player[num_players-1].weapon_model_id = atoi(param[9]);
player[num_players-1].anim_frame = atoi(param[10]);
player[num_players-1].active = 1;
strncpy(connection[con].name, param[1], MAXLENNAME);
player[num_players-1].name = connection[con].name;
generateId(con);
player[num_players-1].global_id = global_object_id_counter-1;
announce(connectEvent,(void*) con);
connection[con].state = Talking;
connection[con].bufferLen = 0;

```

```

dumpStatus(con, num_players-1);

    }
}

/*mv:ID:x:y:z:vx:vy:vz:anim */
if (!strncmp(param[0], "mv", 2))
{
    ply = -1;
    for(k=0; k<num_players; k++)
if (player[k].global_id == atoi(param[1]))
    ply = k;
    if (ply == -1) { fprintf(LOG, "Player doesn't exist!\n"); } else {

player[ply].pos.x = atof(param[2]);
player[ply].pos.y = atof(param[3]);
player[ply].pos.z = atof(param[4]);
player[ply].view.x = atof(param[5]);
player[ply].view.y = atof(param[6]);
player[ply].view.z = atof(param[7]);
player[ply].anim_frame = atoi(param[8]);
announce(moveEvent, (void*) ply);
    }
    connection[con].bufferLen=0;
}

/* Fire */
if (!strncmp(param[0], "fr", 2))
{
    ply = -1;
    for(k=0; k<num_players; k++)
if (player[k].global_id == atoi(param[1]))
    ply = k;
    if (ply == -1) { fprintf(LOG, "Player doesn't exist!\n"); } else {
projectile.plyr_id = atoi(param[1]);
projectile.sx = atof(param[2]);
projectile.sy = atof(param[3]);
projectile.sz = atof(param[4]);
projectile.dx = atof(param[5]);
projectile.dy = atof(param[6]);

```

```

projectile.dz = atof(param[7]);
projectile.weapon = atoi(param[9]);
projectile.speed = atof(param[8]);
announce(fireEvent,(void*) &projectile);
    }
    connection[con].bufferLen=0;
}

if (!strncmp(param[0],"hit",3))
{
    projectile.plyr_id = atoi(param[2]);
    projectile.hit_plyr_id = atoi(param[1]);
    projectile.weapon = atoi(param[3]);
    projectile.dx = atof(param[4]);
    projectile.dy = atof(param[5]);
    projectile.dz = atof(param[6]);
    announce(hitEvent,(void*) &projectile);
    connection[con].bufferLen=0;
}

/* bye: */
if (!strncmp(param[0],"bye",3))
{
    ply=-1;
    connection[con].state = NotConnected;
    fprintf(LOG,"Removing client(s) on filedescriptor %d\n",connection[con].fd);
    connection[con].bufferLen = 0;
    write(connection[con].fd,
"bye\0",
4);
    for(k=0; k<num_players; k++)
if (player[k].global_id == atoi(param[1])) ply=k;
    if (ply == -1) { fprintf(LOG,"Non-existent player id\n"); }
    else {
player[ply].active = 0;
announce(disconnectEvent,(void*) atoi(param[1]));
    }
}
}

```

```

/* die: */
if (!strcmp(param[0],"die",3))
{
    connection[con].state = NotConnected;
    fprintf(LOG,"Client on %d dying!\n",connection[con].fd);
    connection[con].bufferLen=0;
    close(connection[con].fd);
}

if(!strcmp(param[0],"sc",2))
{
    for(k=0; k<num_players; k++)
if (player[k].global_id == atoi(param[1])) ply=k;
    if (ply == -1) { fprintf(LOG,"Non-existent player id\n"); }
    player[ply].score++;
    announce(scoreEvent,(void*) atoi(param[1]));
    connection[con].bufferLen=0;
}

/* say:something */
if (!strcmp(param[0],"say",3))
{
    announcement.connection = &connection[con];
    announcement.message = param[1];
    announce(sayEvent, (void*) &announcement);
    connection[con].bufferLen = 0;
}

/* msg:something */
if (!strcmp(param[0],"msg",3))
{
    announce(genericEvent, (void*) param[1]);
    connection[con].bufferLen = 0;
}

connection[con].bufferLen = 0;

commandsplit_Free(param,params);
}

```

```

/*****
/* Add a connection */

void addConnection(int clientSocketFD, struct sockaddr_in *clientAddress)
{
    int emptyCon = 0;
    fprintf(LOG,"Adding client on filedescriptor %d\n",clientSocketFD);

    while(connection[emptyCon].state!=NotConnected && emptyCon<=MAXCON)
        emptyCon++;
    if (emptyCon > MAXCON) {
        fprintf(LOG,"Too many people connected, but what the hell...\n");
        /* Too many people connected, worry about it later */
    } else {
        connection[emptyCon].state = LoggingIn;
        connection[emptyCon].fd = clientSocketFD;
        connection[emptyCon].hostIP = ntohl(clientAddress->sin_addr.s_addr);
        connection[emptyCon].bufferLen = 0;

        num_players++;
        if ((player = (PLAYER*) realloc(player, sizeof(PLAYER)*num_players)) == NULL)
            { fprintf(LOG,"addConnection(): Out of memory when realloc'ing player\n");
              exit(1);
            }

    }
}

/*****
/* The main loop of the program. Basically sits there waiting for stuff */
/* to happen. Nasty bit around the select to deal with SIGALRMs and */
/* EINTR from select() */

void mainLoop(int serverSocketFD)
{
    /* Variables */
    int result;
    int clientSocketFD;
    int clientLength;
    struct sockaddr_in clientAddress;

```

```
/* Go forever. Well, nearly. */

while(1) {
    /* More variables */
    char ch;
    int con;
    int nRead, highestFD=0;
    //struct hostent *host;
    fd_set FDSset;
    //    const char* temp_string;

    /* Build an FD set so we can listen for activity */
    FD_ZERO(&FDSset);
    for(con=0; con<MAXCON; con++) {
        if(connection[con].state != NotConnected) {
            FD_SET(connection[con].fd,&FDSset);
            if(connection[con].fd > highestFD)
                highestFD = connection[con].fd;
        }
    }

    /* Block and wait for something to happen. select() will return in
       one of three cases:
       * Activity on one of the FDs
       * Timeout (not happening here)
       * If a signal is sent to the process and the signal handler function
         returns normally without exit()ing
       We need to deal with this last case.
    */
    result = select(highestFD+1, &FDSset, NULL, NULL, NULL);

    /* If select returned with an error, check to see if it was because we
       got a server tick */
    if (result<0) {
        if (sigAlrmCaught==true)
        {
            /* If so, ignore it, and reset the sigAlrmCaught */
            sigAlrmCaught=false;
        }
    }
}
```

```
        else
    {
        /* If not, then exit giving a reason. This probably shouldn't
           happen under normal circumstances :) */
        perror("server");
        exit(1);
    }

    }

    if (result>0) { /* result will never == 0 */
        /* Activity detected! Yay! */

        /* Find out which file descriptor it's on... */
        for(con=0; con<MAXCON; con++) {
if (connection[con].state != NotConnected &&
    FD_ISSET(connection[con].fd,&FDSet)) {

        /* If it's on the server socket, then this is a request for a new
           connection, so get the Socket filedescriptor and add the connection
           to our table. */
        if (connection[con].fd == serverSocketFD)
            {
                clientSocketFD = accept(serverSocketFD,
                    (struct sockaddr*) &clientAddress,
                    &clientLength);
                addConnection(clientSocketFD, &clientAddress);
            }

        /* Otherwise it's either activity or something dying */
        else
            {
                /* Try and read some stuff from the socket */
                ioctl(connection[con].fd, FIONREAD, &nRead);

                /* Zero bytes read == connection closing */
                if (nRead==0) {
removeConnection(con);
                }

                /* otherwise we've received some data. Deal with that */
```

```
        else {
read(connection[con].fd, &ch, 1);
/* If it's the termination character, then deal with
the command */
if (ch=='\x00') {
    connection[con].buffer[connection[con].bufferLen] = '\0';
    dealWithCommand(con);
}
/* Otherwise just add it to the incoming buffer.
Gotta love them compound statements :) */
else {
    connection[con].buffer[connection[con].bufferLen++] = ch; }
    }
    }
    }
    }
}

void processArgs(int argc, char* argv[])
{
    int arg;
    for(arg=0; arg<argc; arg++) {
        if (!strncmp(argv[arg], "-p", 2))
            { port=atoi(argv[arg+1]); }
        if (!strncmp(argv[arg], "-v", 2))
            { LOG=stdout; }
    }

    if (LOG==NULL)
        LOG=fopen("/dev/null", "w");
}

int main(int argc, char* argv[])
{
    int serverSocketFD;

    /* Announce */
    printf("\nmxserver version %.2f (Compiled: %s - %s)\n",
```

```
        mxserver_version, __DATE__, __TIME__);
printf("By Chris Whitworth - Part of 'MAVERIK DOOM' by");
printf("Chris Whitworth & Matt Craven\n");
printf("(c)1999 The University Of Manchester\n\n");

/* Setup the shutdown code... */
atexit(shutdn);
signal(SIGQUIT, signalHandler);
signal(SIGTERM, signalHandler);
signal(SIGHUP, signalHandler);

signal(SIGALRM, timerEvent);

processArgs(argc, argv);
serverSocketFD = startServer();
startTimer();

mainLoop(serverSocketFD);
exit(0);
}
```

Appendix B

mxserver.h source

```
/* mxserver.h */

#ifndef __MXSERVER_H__
#define __MXSERVER_H__

#include "networkIncludes.h"

#define mxserver_version 0.11

#define MAXLENNAME    256
#define MAXCON        30
#define MAXBUFFERSIZE 4096
#define TERMINATOR    0x00

/*****
/* Datatypes. */

/* Struct for holding the connections information */
typedef struct {
    enum {NotConnected, LoggingIn, Talking, Server} state;
        /* State of this connection - server also has an entry in the
        connection table so needs its own special state */
    int fd;
        /* Filedescriptor for this connection. Used to build an fdset
        for select() */
    int hostIP;
```

```
        /* IP address of the host as a 32bit (hopefully!) int */
char name[MAXLENNAME];
        /* Nickname of the person connecting */
char buffer[MAXBUFFERSIZE];
        /* Stuff that comes in on a connection gets buffered here until we receive
        a termination character */
int bufferLen;
        /* Amount of data in input buffer */
} Connection;

/* Struct for a message */
typedef struct {
    char* message;
        /* String of message */
    Connection* connection;
        /* Originating connection */
} Message;

typedef struct {
    long int plyr_id, hit_plyr_id;
    float sx, sy, sz;
    float dx, dy, dz;
    int weapon;
    float speed;
} ProjectileDetails;

/* Server message types */
typedef enum {connectEvent,
    disconnectEvent,
    timeEvent,
    genericEvent,
    sayEvent,
    moveEvent,
    fireEvent,
    scoreEvent,
    hitEvent} ServerEvent;

#endif
```

Appendix C

networkclient.c source

```
/* Client calls */
/* Supports polling... */

#include "networkclient.h"
#include "console.h"
#include "string.h"
#include "player.h"
#include "commandsplit.h"
#include "types.h"
#include "weapons.h"

#include <math.h>

extern Connection myConnection;
extern int result;
extern PLAYER* player;
extern int player_my_id;

extern void w_railgun_move(WEAPON_PROJECTILE*);
extern void w_rocket_move(WEAPON_PROJECTILE*);
extern void w_pistol_collide(WEAPON_PROJECTILE*);
extern void w_shotgun_collide(WEAPON_PROJECTILE*);
extern void w_rocket_collide(WEAPON_PROJECTILE*);
extern WEAPON_PROJECTILE* weapon_projectile;
extern void weapons_player_hurt(int, int, MAV_vector*, int);
```

```

PLAYER networkclient_player;

int errno;
int networkclient_NumCallbacks;
NETWORK_CALLBACK* networkclient_Callback;

char temp_buffer[1024];

unsigned int networkclientState=0;

float networkclient_movementThreshold;
float networkclient_rotateThreshold;

/*****
/* Initialise */

void networkclient_init()
{
    printf("\nNetwork client version %.2f (Compiled: %s - %s)\n",
           networkclient_version, __DATE__, __TIME__);
    printf("By Chris Whitworth - Part of 'MAVERIK DOOM'");
    printf(" by Chris Whitworth & Matt Craven\n");
    printf("(c)1999 The University Of Manchester\n\n");
    networkclient_NumCallbacks = 0;
    networkclient_Callback = NULL;

    networkclient_movementThreshold = 1.0;
    /* These two values will be adjusted by an intelligent routine */
    networkclient_rotateThreshold = 0.1;
    /* based on ping time from server */
    networkclient_addCallback("CON",networkclient_cb_connect);
    networkclient_addCallback("ID",networkclient_cb_id);
    networkclient_addCallback("SAY",networkclient_cb_say);
    networkclient_addCallback("MV",networkclient_cb_move);
    networkclient_addCallback("FR",networkclient_cb_fire);
    networkclient_addCallback("DIS",networkclient_cb_bye);
    networkclient_addCallback("HIT",networkclient_cb_hit);
    networkclient_addCallback("MSG",networkclient_cb_msg);
    networkclient_addCallback("SC",networkclient_cb_sc);
}

```

```

/*****
/* Register a new callback                                     */

void networkclient_addCallback(char* command, NETWORK_FUNC functn)
{
    networkclient_NumCallbacks++;
    networkclient_Callback =
        realloc(networkclient_Callback,
            sizeof(NETWORK_CALLBACK)*networkclient_NumCallbacks);
    networkclient_Callback[networkclient_NumCallbacks-1].command =
        (char *) malloc(sizeof(char) * strlen(command));
    sprintf(networkclient_Callback[networkclient_NumCallbacks-1].command,command);
    networkclient_Callback[networkclient_NumCallbacks-1].func = functn;
}

/*****
/* Send a command                                           */

void networkclient_sendCommand(char* com, char** params, int no_params)
{
    char myTemp[1024];
    int command,myResult;
    sprintf(myTemp,"%s",com);
    for(command = 0; command<no_params; command++)
        { strcat(myTemp,":");
          strcat(myTemp, params[command]);
        }
    strcat(myTemp,"\x00");
    if ((myResult=write(myConnection.fd,myTemp,strlen(myTemp)+1))== -1)
        { perror("networkclient_sendCommand"); exit(1); }
}

/*****
/* Open a connection to the specified server                 */

int networkclient_openConnection(char *host, int port, Connection *connection)
{
    struct sockaddr_in serverAddress;
    struct hostent *hostInfo;

```

```

int serverSocketFD, serverLen;

/* Get host info/check host exists */
if ((hostInfo = gethostbyname(host))==0) {
    return EOC_NSH;
}

/* Set up a socket */
serverSocketFD = socket(AF_INET, SOCK_STREAM, 0);
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(port);
serverAddress.sin_addr = *(struct in_addr *)* hostInfo->h_addr_list;
serverLen = sizeof(serverAddress);

/* Connect socket */
if (connect(serverSocketFD, (struct sockaddr*) &serverAddress, serverLen) == -1)
{
    return EOC_COC;
}

/* Set the connection as non-blocking (so read operations will return
immediately) */
fcntl(serverSocketFD, F_SETFL, O_NONBLOCK);

/* Set some status stuff */
connection->fd = serverSocketFD;
connection->state = LoggingIn;
connection->bufferLen = 0;

printf("Logging player in... \n");
networkclient_playerLogin(connection, player_my_id);
return 0;
}

/*****
/* Log a player in. Now, the theory is that this should let us log
multiple players into a single connection (--> bots) but it remains
untested fully... */

int networkclient_playerLogin(Connection* connection, int playerId) {

```

```

char* commandList[1];

commandList[0] = temp_buffer;

/* Send a login command */
sprintf(commandList[0], "%s:%f:%f:%f:%f:%f:%f:%i:%i:%i",
connection->name,
player[playerId].pos.x,
player[playerId].pos.y,
player[playerId].pos.z,
player[playerId].view.x,
player[playerId].view.y,
player[playerId].view.z,
player[playerId].model_id,
player[playerId].weapon_model_id,
player[playerId].anim_frame);
networkclient_sendCommand("hlo", commandList, 1);

/* Initialise our local copy of the player */
networkclient_player.pos.x = player[playerId].pos.x;
networkclient_player.pos.y = player[playerId].pos.y;
networkclient_player.pos.z = player[playerId].pos.z;
networkclient_player.view.x = player[playerId].view.x;
networkclient_player.view.y = player[playerId].view.y;
networkclient_player.view.z = player[playerId].view.z;
networkclient_player.anim_frame = player[playerId].anim_frame;

networkclientState = networkclientState | WAITING_ON_ID;

return 0;
}

/*****
/* Close a connection to the server */

int networkclient_closeConnection(Connection* connection)
{
char *args[1];
args[0] = temp_buffer; sprintf(temp_buffer, "%i", player[player_my_id].global_id);
networkclient_sendCommand("bye", args, 1);

```

```

    connection->state = NotConnected;
    /*read(connection->fd,buf,4);
    if(strncmp(buf,"bye",3)!=0)
    { return 1; }*/ /* This doesn't work at the moment. Never mind */
    close(connection->fd);
    return 0;
}

/*****
/* Announce something */

void networkclient_announce(char* string)
{
    char* args[1];
    args[0] = string;
    if (myConnection.state != NotConnected)
        networkclient_sendCommand("msg",args,1);
    else
        console_printf(string);
}

/*****
void networkclient_issueMove()
{
    char *args[1];
    PLAYER* ply;

    if (myConnection.state != NotConnected) {

        ply = &(player[player_my_id]);
        if (ply->global_id != 0) {
            args[0] = temp_buffer;
            sprintf(temp_buffer,"%i:%f:%f:%f:%f:%f:%f:%i",
                ply->global_id,
                ply->pos.x, ply->pos.y, ply->pos.z,
                ply->view.x, ply->view.y, ply->view.z,
                ply->anim_frame);
            networkclient_sendCommand("mv",args,1);
        }
    }
}

```

```

}

/*****
void networkclient_checkMove()
{
    float movedd,viewdd;

    movedd = sqrt( pow(player[player_my_id].pos.x -
        networkclient_player.pos.x, 2.0)+
    pow(player[player_my_id].pos.y -
        networkclient_player.pos.y, 2.0)+
    pow(player[player_my_id].pos.z -
        networkclient_player.pos.z, 2.0));

    /* Nasty. Should do something with angles if I have time */
    viewdd = sqrt( pow(player[player_my_id].view.x -
        networkclient_player.view.x, 2.0)+
    pow(player[player_my_id].view.y -
        networkclient_player.view.y, 2.0)+
    pow(player[player_my_id].view.z -
        networkclient_player.view.z, 2.0));

    if (movedd > networkclient_movementThreshold ||
        viewdd > networkclient_rotateThreshold) {
        networkclient_issueMove();
        networkclient_player.pos.x = player[player_my_id].pos.x;
        networkclient_player.pos.y = player[player_my_id].pos.y;
        networkclient_player.pos.z = player[player_my_id].pos.z;
        networkclient_player.view.x = player[player_my_id].view.x;
        networkclient_player.view.y = player[player_my_id].view.y;
        networkclient_player.view.z = player[player_my_id].view.z;
        networkclient_player.anim_frame = player[player_my_id].anim_frame;
    }
}

/*****
/* Issue a 'weapon fired' thingy */

void networkclient_issueFire(PLAYER* plyr, int weapon_num, float speed)
{

```



```

&connection->buffer[connection->bufferLen],
1);

/* Nothing to report, return 0 */
if (myResult==-1)
    { if (errno==EAGAIN)
      { return 0; }
    else
      { perror("networkclient_readNetwork"); exit(1); }
    }

/* Oops, can't actually do this... If we've got loads of stuff happening,
we could miss something by reading beyond the end of the buffer, which
would be bad.
Dontcha just *love* asynchronicity?

myResult = read(connection->fd,
connection->buffer+(sizeof(char)*(1+connection->bufferLen)),
MAXBUFFERSIZE); */

/* Instead, this will read until the end of the current command, preserving
any pending commands in the input buffer if there are any */

while (connection->buffer[connection->bufferLen] != '\x00' && myResult!=-1)
    {
        connection->bufferLen++;
        if (connection->bufferLen > MAXBUFFERSIZE)
            {
                connection->bufferLen = MAXBUFFERSIZE;
            }
    }

    myResult = read(connection->fd,
&connection->buffer[connection->bufferLen],
1);
}

/* If we reached the end of the input stream before completing a command,
the terminate here and get some more next poll */
if (myResult==-1)
    { if (errno==EAGAIN)

```

```

        { printf("Incomplete command received...\n");return 0; }
    else
        { perror("networkclient_readNetwork"); exit(1); }
    }

/* Otherwise set the last character of the string to \0 and return */

    connection->buffer[connection->bufferLen] = '\x00';
    return connection->bufferLen;
}

/*****
/* Parse a command if we've got one                                     */

void networkclient_parseCommand(Connection* connection)
{
    char* buffer = connection->buffer;
    int commands;
    char** param = commandsplit_Split(buffer,':', &commands);
    int cb=0;
    for(cb=0; cb < networkclient_NumCallbacks; cb++)
        if (!strcmp(networkclient_Callback[cb].command,param[0]))
            { networkclient_Callback[cb].func(commands,param); }

    connection->bufferLen = 0;
}

void networkclient_cleanUp(Connection* connection)
{
    networkclient_closeConnection(connection);
}

/*****
/* Network command callbacks                                         */

void networkclient_cb_connect(int commands, char** param)
{
    bool reconnecting = false, me = false;
    int plyr=-1;
    int loopvar;

```

```

    MAV_vector pos;

    printf("Connection at %d by %s from %s",
    atoi(param[1]),
    param[3],
    param[2]);

    for(loopvar=0; loopvar<num_players; loopvar++)
    {
        if (player[loopvar].global_id == atoi(param[4]))
    {
        if (loopvar == player_my_id) { me=true; }
        else { plyr = loopvar; reconnecting = true; }
    }
    }

    if (!me) {
        if (!reconnecting) {
            sprintf(temp_buffer,"Connection at %d by %s from %x",
            atoi(param[1]),
            param[3],
            atoi(param[2]));
            pos.x = atof(param[5]);
            pos.y = atof(param[6]);
            pos.z = atof(param[7]);
            player_create(0,0,pos,param[3], NULL, atoi(param[2]));
            plyr = num_players-1; }
        else
        {
            sprintf(temp_buffer,"Reconnection at %d by %s", atoi(param[1]), param[3]);
            pos.x = atof(param[5]);
            pos.y = atof(param[6]);
            pos.z = atof(param[7]);
            player_set_position(&player[plyr], pos);
        }

        player[plyr].model_id = atoi(param[12]);
        player[plyr].weapon_model_id = atoi(param[13]);
        player[plyr].global_id = atoi(param[4]);
        player[plyr].view.x = atof(param[8]);

```

```

        player[plyr].view.y = atof(param[9]);
        player[plyr].view.z = atof(param[10]);
        player[plyr].anim_frame = atoi(param[11]);
        console_printf(temp_buffer);
    }
}

void networkclient_cb_say(int commands, char** param)
{
    char temp_buffer[1024];
    if (commands!=4) { printf("Malformed command - say"); }
    else {
        sprintf(temp_buffer,"%s: %s",param[2],param[3]);
        console_printf(temp_buffer);
    }
}

void networkclient_cb_msg(int commands, char** param)
{
    console_printf(param[1]);
}

void networkclient_cb_id(int commands, char** param)
{
    if (!(networkclientState && WAITING_ON_ID))
        { printf("Erk, unexpected ID received... Ignoring. \n"); }
    else
        { /* For the minute, we only receive an object ID when connecting... */
          player[player_my_id].global_id = atoi(param[1]);
          networkclientState = networkclientState & ~WAITING_ON_ID;
        }
}

void networkclient_cb_move(int commands, char** param)
{ /* MV:id:x:y:z:xv:yv:zv:anim */
    int k;
    int plyr = -1;
    long int id = atoi(param[1]);
    PLAYER_XYZ pos;

```

```

for(k = 0; k < num_players; k++)
    if (player[k].global_id == id) plyr = k;

if (plyr != -1 && plyr != player_my_id)
{
    player[plyr].pos.x = atof(param[2]);
    player[plyr].pos.y = atof(param[3]);
    player[plyr].pos.z = atof(param[4]);
    player[plyr].anim_frame = atoi(param[8]);
    pos.x = atof(param[5]);
    pos.y = atof(param[6]);
    pos.z = atof(param[7]);
    player_set_view(&player[plyr],pos);
}
}

void networkclient_cb_fire(int commands, char** param)
{
    WEAPON_VECTOR pos, dir;
    WEAPON_PROJECTILE_MOVEMENT_FUNC moveFn=NULL, collideFn=NULL;
    int k, plyr=-1;
    long int id = atoi(param[1]);

    for(k = 0; k < num_players; k++)
        if (player[k].global_id == id) plyr = k;

    if (plyr != -1 && plyr != player_my_id)
    {
        pos.x = atof(param[2]);
        pos.y = atof(param[3]);
        pos.z = atof(param[4]);
        dir.x = atof(param[5]);
        dir.y = atof(param[6]);
        dir.z = atof(param[7]);
        switch (atoi(param[9])) {
            case 0: collideFn = w_pistol_collide; break;
            case 1: collideFn = w_shotgun_collide; break;
            case 2: collideFn = w_pistol_collide; break;
            case 3: moveFn = w_rocket_move;
        }
        collideFn = w_rocket_collide; break;
    }
}

```

```

        case 4: moveFn = w_railgun_move;
collideFn = w_pistol_collide; break;
    }
    weapon_projectile_create(pos, dir, atof(param[8]),
                            moveFn, collideFn, atoi(param[9]),plyr);
    }
}

void networkclient_cb_hit(int commands, char** param)
{
    MAV_vector dir;
    int hit_plyr=-1, firing_plyr=-1,k;
    for(k=0; k<num_players; k++)
    {
        if (player[k].global_id == atoi(param[1])) firing_plyr = k;
        if (player[k].global_id == atoi(param[2])) hit_plyr = k;
    }

    if (hit_plyr==player_my_id)
    {
        dir.x = atof(param[4]);
        dir.y = atof(param[5]);
        dir.z = atof(param[6]);
        weapons_player_hurt(hit_plyr, firing_plyr, &dir, atoi(param[3]));
    }
}

void networkclient_cb_sc(int commands, char** param)
{
    int k,plyr=-1;
    for(k = 0; k < num_players; k++)
        if (player[k].global_id == atoi(param[1])) plyr = k;
    player[plyr].score++;
}

void networkclient_cb_bye(int commands, char** param)
{
    int k,plyr=-1;
    char temp_buffer[1024];
    for(k = 0; k < num_players; k++)

```

```

        if (player[k].global_id == atoi(param[1])) plyr = k;
        sprintf(temp_buffer,"%s has disconnected",player[plyr].name);
        console_printf(temp_buffer);
        player_delete(&player[plyr]);
    }

/*****
/* Console Callbacks */

void console_openConnection(int argc, char **argv)
{
    if(argc !=3)
        { console_printf("Syntax: connect [host] [port]"); }
    else
        {
            if ((result=networkclient_openConnection(argv[1],atoi(argv[2]),
                                                    &myConnection))
            {
                switch (result) {
                case EOC_NSH: console_printf("No such host"); break;
                case EOC_COC: console_printf("Can't open connection"); break;
                }
            } else {
                console_printf("Connected to server");
            }
        }
}

void console_closeConnection(int argc, char **argv)
{
    if(argc!=1)
        { console_printf("Syntax: bye"); }
    else
        {
            if (!networkclient_closeConnection(&myConnection))
            {
                printf("Disconnected...\n");
                console_printf("Disconnected from server.");
            }
        }
    else

```

```
{
  console_printf("Disconnect failed for some reason!");
}
}

void console_say(int argc, char **argv)
{
  if(argc!=2)
    { console_printf("Syntax: say \"[whatever]\""); }
  else
    {
      char* coms[1]; coms[0] = argv[1];
      networkclient_sendCommand("say",coms,1);
    }
}
```

Appendix D

networkclient.h source

```
/* client.h */

#ifndef __NETWORKCLIENT_H__
#define __NETWORKCLIENT_H__

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include "player.h"

#include "mxserver.h" /* It's got some useful stuff in, after all. */

#define EOC_NSH 1 /* Error openConnection - No such host */
#define EOC_COC 2 /* Error openConnection - Can't open connection */

#define networkclient_version 0.10

#define WAITING_ON_ID 1

typedef void(*NETWORK_FUNC)(int, char**);

typedef struct {
    char *command;
    NETWORK_FUNC func;
};
```

```
} NETWORK_CALLBACK;

extern void networkclient_init();
extern void networkclient_cleanUp();
extern int networkclient_openConnection(char*, int, Connection*);
extern int networkclient_closeConnection(Connection*);
extern int networkclient_pollNetwork(Connection *);
extern void networkclient_parseCommand(Connection *);
extern int networkclient_playerLogin(Connection *,int);
extern void networkclient_addCallback(char *command, NETWORK_FUNC func);
extern void networkclient_checkMove();
extern void networkclient_issueMove();
extern void networkclient_issueFire(PLAYER*, int, float);
extern void networkclient_announceHit(int, int);
extern void networkclient_announce(char*);
extern void networkclient_sendCommand(char*, char**, int);

extern void networkclient_cb_connect(int, char**);
extern void networkclient_cb_say(int, char**);
extern void networkclient_cb_id(int, char**);
extern void networkclient_cb_move(int, char**);
extern void networkclient_cb_fire(int, char**);
extern void networkclient_cb_bye(int, char**);
extern void networkclient_cb_hit(int, char**);
extern void networkclient_cb_msg(int, char**);
extern void networkclient_cb_sc(int, char**);

extern void console_openConnection(int, char**);
extern void console_closeConnection(int, char**);
extern void console_say(int, char**);

#endif
```

Appendix E

networkincludes.h source

```
/* Quick start network includes header */

#ifndef __NETWORKINCLUDES_H__
#define __NETWORKINCLUDES_H__

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <fcntl.h>

#endif
```

Appendix F

taunt.c source

This was a silly little feature I added to the game whilst frustrated one day. It performs no useful function other than to shout something rude across the network when you kill someone.

```
/* Taunt system */

#include "networkclient.h"
#include "console.h"
#include "taunt.h"

char* taunt_message[TAUNTS] = {
    "Eat that!",
    "Bite me!",
    "Die, loser!",
    "That had to hurt",
    "Sucker!",
    "Come back when you can aim!",
    "Try using a real gun, loser!",
    "I rule!" };

void taunt_taunt(int which)
{
    char* args[1];
    args[0] = taunt_message[which];
    networkclient_sendCommand("say",args,1);
}
```

```
void taunt_random()
{
    int taunt = rand()%TAUNTS;
    taunt_taunt(taunt);
}

void taunt_tauntCallback(int argv, char** argc)
{
    if (argv!=1) { console_printf("Syntax: taunt"); }
    taunt_random();
}
```

Appendix G

taunt.h source

```
/* taunts.h */  
  
#define TAUNTS 8  
  
extern void taunt_taunt(int);  
extern void taunt_random();  
extern void taunt_tauntCallback(int, char**);
```

Appendix H

commandsplit.c source

```
/* Handy utility function to split a command and return it */

#include "commandsplit.h"

char** commandsplit_Split(char* command, char splitChar, int* coms)
{
    int commands=1, commandLen=0;
    int offset, commandNo;
    char** section;

    for (offset=0; offset<strlen(command); offset++)
        if (command[offset]==splitChar)
            commands++;

    section = (char**) malloc(commands * sizeof(char*));
    commandLen=0;
    commandNo=0;
    for (offset=0; offset<strlen(command); offset++)
        if (command[offset]==splitChar)
            { section[commandNo] = (char*) malloc((1+commandLen)*sizeof(char*));
              commandNo++; commandLen=0; }
        else
            commandLen++;

    section[commandNo] = (char*) malloc((1+commandLen)*sizeof(char*));
```

```
commandLen=0;
commandNo=0;

for (offset=0; offset<strlen(command); offset++)
{
    if (command[offset]==splitChar)
        { section[commandNo][commandLen] = '\\0';
          commandNo++; commandLen=0;
        }
    else
        { section[commandNo][commandLen++] = command[offset]; }
}

section[commandNo][commandLen] = '\\0';

*coms = commands;
return section;
}

void commandsplit_Free(char** split, int commands)
{
    int com;
    for (com=0; com<commands; com++)
        free(split[com]);
    free(split);
}
```

Appendix I

commandsplit.h source

```
#ifndef __COMMANDSPLIT_H__
#define __COMMANDSPLIT_H__

#include <string.h>
#include <stdlib.h>

extern char** commandsplit_Split(char*, char, int*);
extern void commandsplit_Free(char**, int);

#endif
```

Appendix J

configure perl script source

```
#!/usr/bin/perl

# This is still a bit crap at the moment; it assumes far too much
# There'll be a better version out soon, promise

sub find_in_lsc
{
    my $srch = shift;
    open LDSOCONF, "/etc/ld.so.conf" or die;
    while(<LDSOCONF>) { if (/ $srch/) {close LDSOCONF; return $_; } }
    close LDSOCONF;
    return '';
}

sub find_in_llp
{
    my $srch = shift;
    @LLP = split /:/, $ENV{'LD_LIBRARY_PATH'};
    foreach $path (@LLP) {
        if ($path =~ / $srch/) { return $path; }
    }
    return '';
}

print "Configuring...\n";
```

```

print "Checking for Maverik 5.2...";

$maverik=find_in_lsc("Maverik-5.2\lib");
if ($maverik eq '')
{
    print "Erk!\n -- Cannot find Maverik-5.2/lib in /etc/ld.so.conf\n";
    print "Checking \LD_LIBRARY_PATH...";
    $maverik = find_in_llp("Maverik-5.2");
    if ($maverik eq '') {
        print "Erk!\n -- Can't find Maverik 5.2!\n";
        print "Please enter the path where the Maverik 5.2 libraries and includes";
        print " can be found:\n> ";
        $maverik = <> chomp $maverik;
        open TEMP,"$maverik/lib/libmaverik.so"
            or die "Maverik libraries not found at $maverik/lib/";
        close TEMP;
    }
    else
    { print "OK\n"; }
}
else
{
    print "OK\n";
    $maverik =~ s%(.)lib(.*)%\1\2%;
    chomp $maverik;
}

print "Writing Makefile...\n";

open MAKEFILE,">Makefile" or die "Can't open Makefile for writing";
print MAKEFILE "#Makefile for MavDOOM with network support\n";
print MAKEFILE "#Generated by Chris Whitworth's magick perl configure script\n\n";
print MAKEFILE "MAVINC= $maverik/incl\nMAVLIB= $maverik/lib\n\n";
while(<DATA>) { print MAKEFILE; }
close MAKEFILE;

print "Done!\n";

__END__
CC= gcc

```

```

CFLAGS= -I${MAVINC} -I/usr/include -Wall -s -O3 -finline-functions\
        -fomit-frame-pointer -funroll-loops -mpentium

LIBS= -L${MAVLIB} -L/usr/lib -lmaverik
XLIBS= -L/usr/X11/lib -L/usr/X11R6/lib -lX11 -lXext -lXmu -lXt -lXi
GL_LIBS= ${LIBS} -lglut -lMesaGLU -lMesaGL -lm ${XLIBS}

TARGETS = mavdoom mxserver

all: ${TARGETS}

mxserver: mxserver.o commandsplit.o
${CC} ${CFLAGS} $^ -o $@ ${GL_LIBS}

mxserver.o: mxserver.c
${CC} ${CFLAGS} -c mxserver.c

commandsplit.o: commandsplit.c commandsplit.h
${CC} ${CFLAGS} -c commandsplit.c

cs_test: cs_test.o commandsplit.o
${CC} ${CFLAGS} $^ -o $@

mavdoom: mavdoom.o particle.o copy_callbacks.o navigation.o console.o\
player.o weapons.o mdm.o md2.o image.o menu.o sprite.o networkclient.o\
        commandsplit.o taunt.o
${CC} ${CFLAGS} $^ -o $@ ${GL_LIBS}

mavdoom.o:mavdoom.c particle.h copy_callbacks.h navigation.h player.h\
        console.h weapons.h mdm.h menu.h sprite.h taunt.h
${CC} ${CFLAGS} -c mavdoom.c

particle.o:particle.c particle.h copy_callbacks.h
${CC} ${CFLAGS} -c particle.c

copy_callbacks.o: copy_callbacks.c copy_callbacks.h
${CC} ${CFLAGS} -c copy_callbacks.c

navigation.o: navigation.c navigation.h console.h player.h
${CC} ${CFLAGS} -c navigation.c

```

```
console.o: console.c console.h image.h md2.h mdm.h navigation.h\  
particle.h weapons.h  
${CC} ${CFLAGS} -c console.c  
  
player.o: player.c player.h image.h md2.h  
${CC} ${CFLAGS} -c player.c  
  
weapons.o: weapons.c weapons.h navigation.h  
${CC} ${CFLAGS} -c weapons.c  
  
mdm.o: mdm.c mdm.h  
${CC} ${CFLAGS} -c mdm.c  
  
md2.o: md2.c md2.h image.h  
${CC} ${CFLAGS} -c md2.c  
  
image.o: image.c image.h  
${CC} ${CFLAGS} -c image.c  
  
menu.o: menu.c menu.h image.h md2.h  
${CC} ${CFLAGS} -c menu.c  
  
sprite.o: sprite.c sprite.h player.h image.h  
${CC} ${CFLAGS} -c sprite.c  
  
networkclient.o: networkclient.c networkclient.h mxserver.h networkIncludes.h  
${CC} ${CFLAGS} -c networkclient.c  
  
taunt.o: taunt.c taunt.h networkclient.h  
${CC} ${CFLAGS} -c taunt.c  
  
clean:  
rm -f ${TARGETS} *.o  
  
devclean: clean  
rm -f *~
```