

Quake III Arena Game Structures

D. Stefyn*, A.L. Cricenti, P.A. Branch

Centre for Advanced Internet Architectures, Technical Report 110209A

Swinburne University of Technology

Melbourne, Australia

4009134@swin.edu.au, tcricenti@swin.edu.au, pbranch@swin.edu.au

Abstract—This report presents results from the analysis of the Quake III Arena source code in order to gain insights into how the packets flowing between server and client are constructed. The server to client packets contain game state information that is based on the aggregation of the changes in state of the individual game entities in the vicinity of the player, whilst the client to server packets contain information of the changes in the individual players' state. Ultimately the information sent affects the size of the packet and may be used as a basis for determining network performance characteristics of the game or for creating modifications to the source code. Quake III Arena is seen to employ both delta states to incrementally updates changes in the game from an initial baseline and Huffman coding. These techniques have the effect of minimising the size of packets, and so improving the networking performance. Quake III Arena is programmed largely in the C programming language and heavily employs the use of "C structs" to store information on various items within the game.

Keywords - *Quake III Arena Source Code, FPS game traffic.*

I. INTRODUCTION

Quake III Arena (or Quake III) [1] is a game of the First Person Shooter (FPS) genre, first released in 1999 and still remains very popular today. The name of this game itself can be a point of confusion amongst readers. Quake III Arena, Quake 3, Q3A all refer to the same game which is analysed in this report. An expansion named Quake III: Team Arena was later released, which had a stronger focus on team play [2]. Many other derivatives of this game have been created based on the released source code, including the open source game OpenArena [3].

The Quake III Arena source code has been released under the open source GPL license since 2005 [4], providing a starting point for many amateur game developers. The current huge popularity of games, in particular

*The author was an undergraduate engineering student while writing this report

FPS games has spurred a lot of research into the network traffic produced by these games.

A number of research projects have analysed the traffic produced by Quake III Arena [5], [6] and some other FPS games that are based on the same engine [7]. This report aims to provide researchers with a quick orientation to the Quake III Arena source code, particularly as it relates to the information that the game engine transmits across the network. By having a structured introduction to these game aspects, these characteristics can also be extrapolated for comparing the network performance of other FPS games, whether they are open or closed source.

Quake III Arena was written at a time when dial up modems were still popular and a lot of care was taken to minimise network traffic so as to effectively deal with issues such as congestion and latency[8]. With broadband connections being commonplace these days, some of these factors are of lesser concern today. However, latency is still an important factor in FPS games and many optimisations have been continued through the genre. A delay of half a second can be catastrophic in a FPS and can effectively make or break a game. Basic principles in minimising latency are still the same and often are centred on the transmission of frequent, but minimal, game updates between client and server. Shorter packets keep latency down by minimising serialisation delays, while frequent updates allow for smoother animation of avatar activities (movements and shooting).

A technique used in Quake III Arena to minimise the size of the update packets is to make use of maps with many small interconnected rooms. This feature still keeps the gameplay interesting by ensuring that players interact with each other often, but avoids having large numbers of players on the screen and many simultaneous animations, such as rocket blasts. This technique allows the snapshot update packets to be kept as small as possible, as well as minimising the amount of rendering



Figure 2. Typical View of a Quake III Arena game (from [17])

virtual world, Typically the player holds some form of weapon and navigates a map in the search of enemies to fight. Combat is usually the central focus of the game and most games award the winning score to the player with the highest amount of kills (frags). A variety of weapons and power-ups are available for collection. Power-ups can increase the player’s advantage through increased health and armour protection or allow ability to inflict extra damage. This gameplay summarises the strategy behind Quake III Arena , as can be seen in the screenshot Figure 2. A Heads Up Display (HUD), is applied over the view of the player and shows statistics such as the remaining ammuntion and health [15], [16].

C. Interest Management and Field of View

Given the 3D nature of the game and specific style of gameplay, a number of concepts are worth examining before detailing the source code itself.

As humans, we navigate largely through the aid of our vision. The angular extent of what a person sees at any given moment is called the “field of view” (FOV) [18], [19]. This field of view is determined by the direction that the person faces and is approximately 180 degrees. In an FPS game scenario, the visible area of the virtual world is also referred to as the field of view or sometimes the “aura of interest”, viewpoint and so forth. The game state of a player in Quake III Arena is an aggregate of the player’s current location and actions combined with the focus (area being viewed) and nimbus (viewable footprint) of each individual game entity, (other players, bots etc.).

A visual representation may be better explain these concepts, Figure 3 shows the focus and the nimbus of each player in a game of “Hide and Seek”.

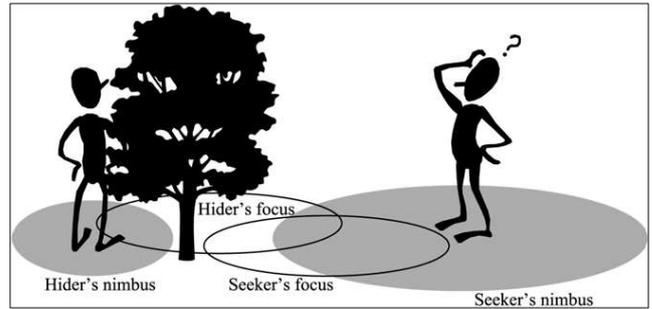


Figure 3. Focus and Nimbus as seen in "Hide and Seek" (from [20])



Figure 4. Changing the Field of View in Quake III Arena (from [21])

In this situation the player auras are asymmetrical as the Hider is able to see into the Seeker’s aura of interest, whilst staying out of view of the Seeker.

The field of view in Quake III Arena is much more symmetrically defined and the cone of vision experienced by each player is largely the same. The FOV can be adjusted either to increase or decrease the field of vision of a player, as shown in Figure 4. This screenshot shows the difference experienced by the player when changing the FOV variable within Quake III Arena . The image on the left shows an extended FOV providing 110 degree vision. The image on the right shows the default game FOV of 90 degree vision. As there was an increase in the usage of wide-screen monitors at the time that Quake III Arena was written, the FOV component allowed for a better experience when utilising a wide-screen monitor.

Due to the amount of extra vision experienced by these players, a gameplay advantage is apparent as the player can see peripheral detail they may otherwise miss. Simply increasing the FOV does however also cause a disadvantage in some cases. Distant objects have a smaller screen footprint than they would for a player using a larger field of view, increasing aiming difficulty.

The FOV can be changed within the game by opening the console and issuing the command: `"/cg_fov 120"`. The example sets a FOV of 120 degrees. True 360 degree vision is not however possible without modification to the game code. Through experimentation, game FOV

appears to be effectively limited to a maximum of 160 degrees of vision.

FOV is also manipulated as part of gameplay when zoom is used, or when the viewpoint is warped, as happens when the Quake III Arena player is underwater.

An area mask is used to calculate what the player can see based on their field of view and comprises the known game state for that player. As the player moves about, the area mask shifts and information from the new FOV is communicated by the game server.

D. Quake III Arena Bots

Before describing the bulk of gameplay in detail, the notion of a bot will be introduced.

Bots are computer players in Quake III Arena and the name is taken from a shortened form of the word robot. A bot has a degree of artificial intelligence, allowing it to behave similarly to a human player within the game. In this way, extra players are always available when needed to populate extra multi-player game slots, or simply to allow for game practice when no other human opponents are available.

The Area Awareness System (AAS) provides a bot with information about the game world. The Quake series of games has had many modifications or 'mods' contributed by the community of players. As such it is important for a bot to possess as much intelligence of its own, so that it can adequately navigate new maps, although assistance is provided through the AAS.

Quake III Arena maps are distributed with the main BSP map file and commonly with an AAS file. The AAS is a pre-compiled representation of the map to increase the efficiency of bots on that map. Pathfinding and reachability information is provided for traversal of the 3-dimensional map. Obstacles impeding standard navigation are optimised and accounted for, such as gaps in floors, swimming areas and teleport points. Fuzzy logic is used to calculate the need to pick up a particular power-up or to re-arm oneself with more ammunition, whilst maintaining the key goal of winning the game (kills, frags etc) [22], [23].

A comprehensive paper by van Waveren [24] on the subject of Quake III Arena bots is provided in the references section. This paper also addresses some aspect of the main game, overlapping some of the content provided in this report.

E. Rendering of Game Maps

Due to the intense visual nature of FPS games, a large part of code is devoted to graphics rendering. This

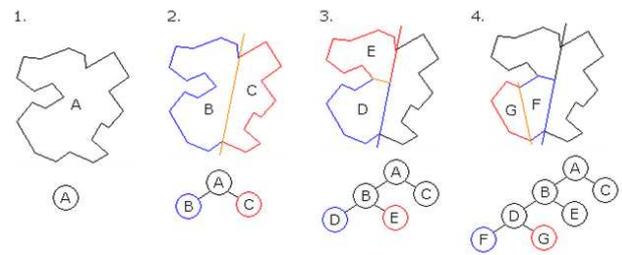


Figure 5. BSP tree generation (from [25])

section is included to assist with concepts used later in the report, such as game portals mentioned when discussing viewable player states. Although little public analysis has centred on the Quake III Arena game engine or network code, the reverse can be said for map design. Significant effort is spent on optimising maps and as such many articles and tutorials are readily available for level designers of this game. A portal is a viewable area between two surfaces. By segmenting viewable game sections into smaller portions, the entire map need not be rendered, and only sections applicable to the player need be presented.

Quake III Arena uses Binary Space Partitioning (BSP)[25] to recursively subdivide spaces for performance and viewing benefits. If there is no activity within a viewable portal, it is essentially ignored until needed.

Referring to Figure 5 a quick depiction of BSP is provided below.

1. A is the root of tree and comprises the entire polygon .
2. A is divided into areas B and C
3. B is divided into areas D and E.
4. D is divided into areas F and G. These are convex and therefore become leaves on the tree.

The end result is a series of leaves which can be rendered. Part of the design process involves ensuring the end result is a closed convex shape. As a leaf comprises an area the player is able to view at any given time, a line of vision from any two points in the leaf should be unobstructed. This is the situation in the convex shape in Figure 6. The concave shape can be seen to provide an obstructed view, which alludes to the impossible situation where a player could see around a corner, which needs to be avoided.

Some acronyms will be heavily used when reading further into BSP and map design. To make these references easier, some key terms are included below[26].

- PVS - Potentially Viewable Set
- BSP - Binary Space Partitioning, a method for

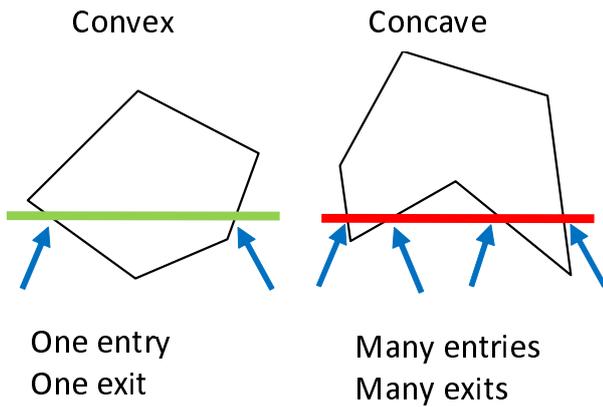


Figure 6. Convex and Concave Leaves (from [26])

recursive subdivision of spaces into planes, for improving rendering efficiency.

- Portals - Viewable areas between surfaces. These limit the areas that need to be rendered at one time. Segmented for small size viewpoints / interactions.
- Leafs – Convex rendering sections which may overlap each other to provide better fits between portals.

III. INITIAL SOURCE CODE ANALYSIS

A. About the Source code

Compilation of the source code requires possession of a retail CD-ROM of the Quake III Arena game. Although the game code was released under the open source GPL license, a number of resources such as textures and maps are required to be copied from the original game media if the original Quake III Arena is to be built. However, in the case of Open Arena all the required components are available at the Open Arena website [3]. Not much detail exists to aid in the analysis of the Quake III Arena game engine or source code, however, some starting points are included as references to this report.

B. File hierarchy and organisation

The Quake III Arena source code combines close to 2000 separate files. For ease of management, it is conveniently divided into separate directories. A brief overview of the directory structure is provided in Table I on page 5. The code is written predominantly in the C programming language with a small number of components also written in C++ or assembly language. The GPL source code for the latest game version can be downloaded from the id Software FTP site [4]. This report was based on version 1.32b of the code. Once the

Table I
PROGRAM FILE LOCATIONS

Directory	Description
<code>\code\game</code>	Hierarchical root for much of the game code.
<code>\code\cgame</code>	Client side implementation of the game code.
<code>\code\qcommon</code>	Common definitions for both client and server. Delta read/writes, Huffman coding and network transmission undertaken here.
<code>\code\client</code>	Client side files.
<code>\code\server</code>	Server side files.
<code>\code\(\macosxlinux\win32)</code>	Operating System specific files. Actual network frame construction tends to happen here.
<code>\code\renderer</code>	Various graphics rendering files.

project has been extracted, a number of directories are evident, with the key ones shown in Table I.

The source code tends to use several general naming conventions throughout. A prefix of 's' will generally denote the server while a prefix of 'c' generally denotes the client.

Quake III Arena is implemented within a virtual machine. This provides an abstraction layer to assist the game in remaining platform independent. The code for the virtual machine and for operating system specific functions can mostly be ignored as it does not contribute many additions of interest to this report. Some low level functionality such as network driver interfacing takes place within OS-specific files, these are not investigated in this report. Differences in low level network functionality are not expected to make much difference in the size of network packets and are therefore left alone. Assembly of data for network transmission takes place at higher levels in the code structure and will be covered in depth.

C. Relationships between game components

A large C software project is usually made up of many smaller files to assist organisation and modularity of the code. The ".c" files contain the bulk of the code, whereas the ".h" files are header files which can be included within another file to reference all the functions defined within the header. By examining the header files of the server and client portions of the code, references to further centralised header files become apparent, as shown in Figure 7.

Several common header files are referenced by both server and client sides of the code. Of interest are

```

//// File: /code/server/server.h: (primary header for
server)
#include "../game/q_shared.h"
#include "../qcommon/qcommon.h"
#include "../game/g_public.h"
#include "../game/bg_public.h"
=====
//// File: /code/client/client.h (primary header for client)
#include "../game/q_shared.h"
#include "../qcommon/qcommon.h"
#include "../renderer/tr_public.h"
#include "../ui/ui_public.h"
#include "../cgame/cg_public.h"
#include "../game/bg_public.h"

```

Figure 7. Header File References

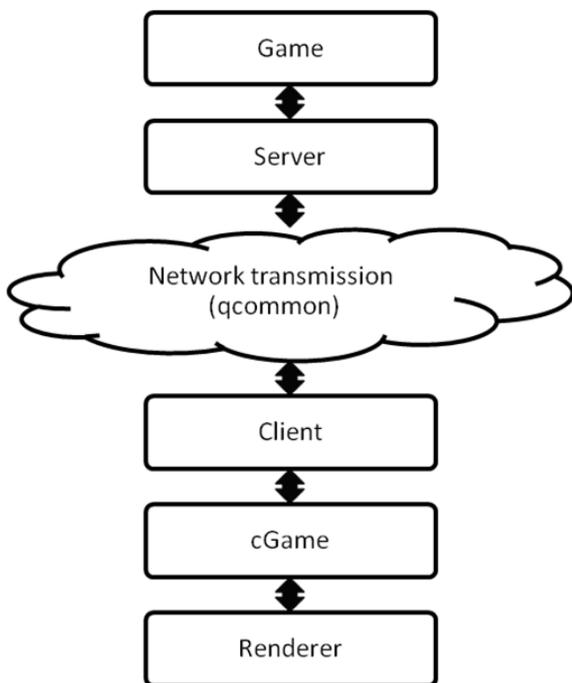


Figure 8. Quake III Arena Program Structure

the ‘game’ and ‘qcommon’ header files respectively. Analysis of other header files in the Quake III Arena code yields similar results, Figure 8 on page 6 shows the relationship between the various parts of the code.

An important aspect is the use of two separate game states. The server maintains its own global representation of the game state. The client also maintains a local representation of the game, *cgame*. Communications are over the network link and the two game states are kept synchronised. Branching out from the *cgame* are the various functions local to the user – functionality

specific to the operating system, graphics rendering, user interface etc.

The structure of the code represented by Figure 8 is not completely accurate in a technical sense, as there is some function sharing between various portions of the game’s code. One example of this is references from the client system to a shared header belonging to game. The file */code/game/bg_public.h* contains shared definitions for both server and client. This could theoretically be segmented further to provide further separation of code, however, the functions within are used by both server and client. For the purposes of visual representation and functionality, Figure 8 provides a good reference for visualising the interaction of the various components of the game code.

IV. SERVER / CLIENT COMMUNICATION

Quake III Arena’s communication model is based on a client server architecture. The message exchange processes between client and server occurs through the definitions supplied in *qcommon.h*. Communication in Quake III Arena is conducted primarily through the exchange of delta states - game updates based on changes in the game from the last acknowledged state. This is a bandwidth saving measure that avoids the need to send complete game state information with each update communication.

A. Delta States

Player states in Quake III Arena are transmitted as incremental updates to a prior game state. An initial baseline state is transmitted when a new map is loaded. Player statistics such as position, orientation and inventory are then periodically transmitted to the server in the form of delta updates. The server also periodically updates the client with its own delta states, containing the status of other players and entities. Since the entire game state is not transmitted every update period, the size of update packets remains comparatively small and network traffic is reduced.

A new delta accounts for any differences in the game from the last acknowledged delta state. In case of lost packets during transmission, a small set of recent delta states are buffered. This allows several game states to be predicted until synchronisation can be re-established. In this way, gameplay can still appear smooth to the end user even when a small amount of packet loss is experienced. If many packets are dropped, the size of the delta update packets would be expected to increase.

```

/// File: /code/qcommon/qcommon.h
// server to client communication
//
enum svc_ops_e {
    svc_bad,
    svc_nop,
    svc_gamestate,
    svc_configstring, // [short] [string] only in
gamestate messages
    svc_baseline, // only in gamestate messages
    svc_serverCommand, // [string] to be executed by
client game module
    svc_download, // [short] size [size bytes]
    svc_snapshot,
    svc_EOF
};

```

Figure 9. Command Sequences (Server to Client)

B. Huffman Coding

Delta states provide incremental updates to a previous game snapshot. This is conducted primarily through the *qcommon* library shared by server and client and is used as the base library for network transmission.

The header file */code/qcommon/qcommon.h* is responsible for much of the network functionality between server and client components. Delta states are generated by individual server and client libraries, but are further processed in *qcommon*. Delta states reduce the size of network transmissions, however, these are further compressed through the use of Huffman coding. The same header file *qcommon.h* oversees this with the file */code/qcommon/msg.c* providing much of the lower level functionality.

C. Server to Client Communication

The code fragment in Figure 9 on page 7 shows the various commands for the communications from the server to the client. *SVC* in this case indicates communication from the server, which the client will then process.

To gain further insight, inspection into another file is needed. Searching the code base for the commands listed in Figure 9 yields the following extra information see Figure 10.

Most of the conditions are handled in the single fragment (Figure 10). It is worth noting the first check is for an error condition to counter corrupt packets. Further conditions can be found elsewhere in the code. These are subsets of the functions given in Figure 10.

Parsing sections of code such as depicted in Figure 11 allows the commands to be analysed individually. The findings are summarised in Table II.

```

//// File: /code/client/cl_parse.c
switch ( cmd ){
default:
    Com_Error (ERR_DROP,"CL_ParseServerMessage:
Illegible server message\n");
    break;
case svc_nop:
    break;
case svc_serverCommand:
    CL_ParseCommandString( msg );
    break;
case svc_gamestate:
    CL_ParseGamestate( msg );
    break;
case svc_snapshot:
    CL_ParseSnapshot( msg );
    break;
case svc_download:
    CL_ParseDownload( msg );
    break;

```

Figure 10. Breakdown of Commands (Server to Client)

```

if ( cmd == svc_configstring ) {
[ ... ]
// append it to the gameState string buffer
cl.gameState.stringOffsets[ i ] =
cl.gameState.dataCount;
Com_Memcpy( cl.gameState.stringData +
cl.gameState.dataCount, s, len + 1 );
cl.gameState.dataCount += len + 1;
} else if ( cmd == svc_baseline ) {
    newnum = MSG_ReadBits( msg, GENTITYNUM_BITS );
    if ( newnum < 0 || newnum >= MAX_GENTITIES ) {
        Com_Error( ERR_DROP, "Baseline number out of
range: %i", newnum );
    }
    Com_Memset ( &nullstate, 0, sizeof( nullstate ));
    es = &cl.entityBaselines[ newnum ];
    MSG_ReadDeltaEntity( msg, &nullstate, es, newnum
);
} else {
    Com_Error( ERR_DROP, "CL_ParseGamestate: bad
command byte" );
}

```

Figure 11. - Network Command Sequences (Server to Client)

D. Client to Server Communication

A similar set of commands can be found for communication from the client to the server.

The main difference in commands here is the use of much smaller data structures to transport information. The client is mainly responsible for updating movement and information about the current player as denoted through *usercmd_t*. Client commands are also able to be sent, these refer to game variables and server commands that are further described in Section IX. Examples of these commands are requesting a ping time or requesting a disconnect from the server. Issues with network communication are also transmitted through the client commands.

Table II
NETWORK COMMAND SEQUENCES (SERVER TO CLIENT)

Server command	Action
svc_bad	Corrupt message, discard
svc_nop	Take no action
svc_gamestate	A new gamestate has been sent. Current gamestate to be wiped, and a new gamestate to be created with given data. This can happen when client delta cache is too old, so a new gamestate has to be resent, or when starting a new game.
svc_configstring	Two sub-commands are used to aid this result
svc_baseline	Locate new commands from server, for use by cgame when needed. Game variables can be found here
svc_serverCommand	Parse (short) download message from server
svc_downloadsvc_snapshotsvc_EOF	Standard end of file indicator to stop parsing a received message

```

//// File: /code/qcommon/qcommon.h
// client to server
enum clc_ops_e {
    clc_bad,
    clc_nop,
    clc_move, // [[usercmd_t]
    clc_moveNoDelta, // [[usercmd_t]
    clc_clientCommand, // [string] message
    clc_EOF
};

```

Figure 12. - Command Sequences (Client to Server)

V. KEY GAME STRUCTURES

C structs and enums store the vast majority of game state information and as such are a very useful starting point for analysing how the game works. An enum can be used to define a custom data type, whereas a struct is used to hold different variables, which may be of different types. A struct may also hold other structs within it and this is commonly done throughout the Quake III Arena code. Many items are often subsets of much larger sets of structures, therefore some recursion is often required to find detail of a specific structure. The variable naming conventions for most part throughout the code are fairly descriptive and therefore easy to follow.

```

//// File: /code/qcommon/qcommon.h
typedef struct {
    qboolean allowoverflow; // if false, do a
    Com_Error
    qboolean overflowed; // set to true if the
    buffer size failed (with allowoverflow set)
    qboolean oob; // set to true if the buffer size
    failed (with allowoverflow set)
    byte *data;
    int maxsize;
    int cursize;
    int readcount;
    int bit; // for bitwise reads and writes
} msg_t;

```

Figure 13. - Network Transport Mechanism (msg_t)

```

//// File: /code/game/q_shared.h
// usercmd_t is sent to the server each client frame
// It is used to communicate actions taken by the
// player, thus used to create game deltas
typedef struct usercmd_s {
    int serverTime;
    int angles[3];
    int buttons;
    byte weapon; // weapon
    signed char forwardmove, rightmove, upmove;
} usercmd_t;

```

Figure 14. - Transmission of user actions via usercmd_t

A. Examination of Structures

C structures are heavily used throughout game's code and several of the key data structures are identified here.

1) *msg_t*: Information for transmission over the network from the server to the client is transported via the *msg_t* struct. Refer to Figure 13.

2) *usercmd_t*: Commands from individual clients are sent to the server through the *usercmd_t* structure. Refer to Figure 14. The information contained within this struct is minimal and consists mainly of user actions. The server progressively updates the game state based on this information.

3) *snapshot_t*: A snapshot is perhaps one of the most important structures in the game and it comprises the state of the game at a particular point in time. The client maintains its own game snapshot which it can use for prediction of game states in the case of lost packets. Regular snapshot are sent from the server to update the client with the view it should have of the game at that point in time.

Worth noting from the snapshot structure in Figure 15 is that the current player state (*playerState_t*) is contained in the variable *ps*, and the entity state is contained in the array of entities. With the definition included in the code fragment, *MAX_ENTITIES_IN_SNAPSHOT* can be seen to limit the ability of the client to track a maximum of 256 entities.

```

//// File: /code/client/client.h
// snapshots are a view of the server at a given
time
#define MAX_ENTITIES_IN_SNAPSHOT 256
// Snapshots are generated at regular time intervals
by the server,
// but they may not be sent if a client's rate level
is exceeded, or
// they may be dropped by the network.
typedef struct {
    int snapFlags;          // SNAPFLAG_RATE_DELAYED,
etc
    int ping;
    int serverTime;        // server time the
message is valid for (in msec)
    byte areamask[MAX_MAP_AREA_BYTES]; // portalarea
visibility bits
    playerState_t ps;      // complete information
about the current player at this time
    int numEntities;      // all of the entities
that need to be presented
    entityState_t entities[MAX_ENTITIES_IN_SNAPSHOT];
// at the time of this snapshot
    int numServerCommands; // text based server
commands to execute when this
    int serverCommandSequence; // snapshot becomes
current
} snapshot_t;

```

Figure 15. - Snapshot Data Structure

This structure can be seen to encapsulate the essential information needed for the game state. As the map is known by both server and client ahead of time, information contained within this struct is simply overlaid on top of the existing map and presented to the game player.

4) *entityState_t* : This encompasses the state of the individual game entities (players, bots etc.) and it is one of the most encompassing structures within the game (see Figure 16). The previously mentioned *playerState_t* is a superset of all *entityState_t* objects within the game and holds the current state of all game action.

Looking at an individual *entityState_t* helps to provide some grounding to reinforce this concept. As will be seen, an entity may be virtually any actionable item in the game: a player (such as a human or bot), missile (rockets, bullets), or a mover (buttons, doors, platforms)

5) *playerState_t* : The structure that contains the most detail after a snapshot is that of a player. Although abbreviated, the excerpt in Figure 17 on page 10 illustrates some of the information stored within a player's state. The information stored in this structure contains the necessary details to calculate the player's view point and direction, weapon and damage information, and whether the player is is running or not.

B. Structures Summary

As these structures will be frequently referenced throughout the game, it is worth including a summary

```

//// File: /code/game/q_shared.h
// entityState_t is the information conveyed from
the server
// in an update message about entities that the
client will
// need to render in some way
// Different eTypes may use the information in
different ways
// The messages are delta compressed, so it doesn't
really matter if
// the structure size is fairly large
typedef struct entityState_s {
    int number; // entity index
    int eType; // entityType_t
    int eFlags;
    trajectory_t pos; // for calculating position
    trajectory_t apos; // for calculating angles
    int groundEntityNum; // -1 = in air
    int constantLight; // r + (g<<8) + (b<<16) +
(intensity<<24)
    int loopSound; // constantly loop this sound
    int solid; // for client side prediction,
trap_linkentity sets this properly
    int event; // impulse events -- muzzle flashes,
footsteps, etc
    int eventParm;
// for players
    int powerups; // bit flags int weapon; //
determines weapon and flash model, etc
    int legsAnim; // mask off ANIM_TOGGLEBIT
    int torsoAnim; // mask off ANIM_TOGGLEBIT
    int generic1;
} entityState_t;

```

Figure 16. - Entity State data structure

Table III
STRUCTURES SUMMARY

Structure	Description
msg_t	Primary method for transporting information from server to client
usercmd_t	Client action updates from client to server
snapshot_t	A snapshot provides a view of the game at a particular time. The server regularly calculates the game state the client should see and relays it through this mechanism for the client to locally store
entityState_t	Object for storing entity information (Entity can be a player, weapon, platform etc)
playerState_t	A subset of an entity, player state holds all information on a player

for later reference, see Table III.

VI. GAME STATE UPDATES

Now that key game structures have been identified, these are able to be put into context for the rest of the game.

On initiation of a game, the server establishes a base-

```

//// File: /code/game/q_shared.h
// playerState_t is the information needed by both the client and
server
// to predict player motion and actions
// playerState_t is a full superset of entityState_t as it is used by
players,
// so if a playerState_t is transmitted, the entityState_t can be
fully derived
// from it.
typedef struct playerState_s {
    int commandTime; // cmd->serverTime of last executed command
    int pm_type;
    int bobCycle; // for view bobbing and footstep generation
    int pm_flags; // ducked, jump_held, etc
    int pm_time;
    vec3_t origin; vec3_t velocity;
    int weaponTime;
    int gravity;
    int speed;
    int delta_angles[3]; // add to command angles to get view
direction
// changed by spawns, rotating objects, and teleporters
    int groundEntityNum; // ENTITYNUM_NONE = in air
    int legsTimer; // don't change low priority animations until this
runs out
    int legsAnim; // mask off ANIM_TOGGLEBIT
    int torsoTimer; // don't change low priority animations until this
runs out
    int torsoAnim; // mask off ANIM_TOGGLEBIT
    int movementDir; // a number 0 to 7 that represents the relative
angle
// of movement to the view angle (axial and diagonals)
// when at rest, the value will remain unchanged
// used to twist the legs during strafing
    vec3_t grapplePoint; // location of grapple to pull towards if
PME_GRAPPLE_PULL
    int eFlags; // copied to entityState_t->eFlags
    int eventSequence; // pmove generated events int
events[MAX_PS_EVENTS];
    int eventParms[MAX_PS_EVENTS];
    int externalEvent; // events set on player from another source int
externalEventParm;
    int externalEventTime;
    int clientNum; // ranges from 0 to MAX_CLIENTS-1
    int weapon; // copied to entityState_t->weapon
    int weaponstate;
    vec3_t viewangles; // for fixed views
    int viewheight;
// damage feedback
    int damageEvent; // when it changes, latch the other parms
    int damageYaw;
    int damagePitch;
    int damageCount;
} playerState_t;

```

Figure 17. - Player State data Structure

line for player and game states. Periodically the server calculates the global game state based on all received deltas from each client. The server then constructs a “snapshot” for each client by adding together information about the change of state of game entities relevant to that client. This snapshot is compressed and transmitted to the respective clients, presenting them with actions of other players and entities within their vicinity.

A. Client to Server Updates

Client to server updates are relatively simple and are much easier to describe.

In Section IV-C communication methods were described between the client and server portions of the game. As noted earlier, commands sent from client to server are much fewer and are mainly involved in transmitting user actions such as movement information and firing of weapons.

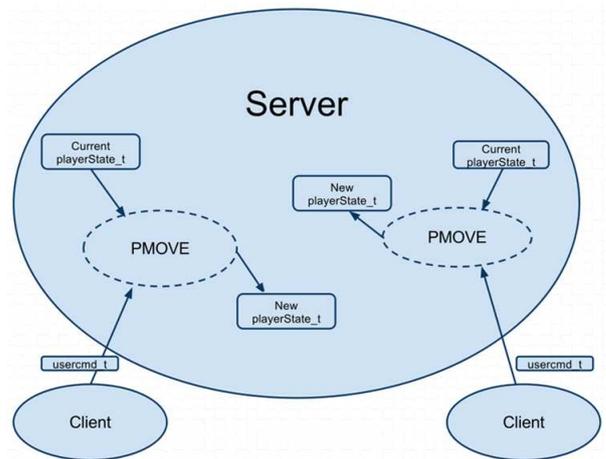


Figure 18. - Communication from Client to Server

The game subsystem described earlier contains the server implementation of the game. It also contains some libraries that are shared between game server and client. These are defined in the header file `\code\game\bg_public.h`. The *PMOVE* module, defined in this library, is the main module for updating player state information throughout a game. Inputs taken are the current player state (*player_state_t*), and the received user actions from the client (*usercmd_t*). A new *player_state_t* is generated that represents the current state of players in the game. To aid readability, this may be presented visually as shown in Figure 18.

B. Server to Client Updates

Packets from the server to the client contain more information than those in the reverse direction. The server periodically creates game snapshots based on the point of view of the client. All visible entities are aggregated into a single snapshot of the player’s game view. This snapshot is delta compressed and Huffman encoded and sent to the client via *msg_t*.

This snapshot is received from the client and then decoded. The client will store a number of these snapshots to assist with delta decoding, client side prediction and so forth.

VII. SNAPSHOTS

Snapshots are an important part of the game as they hold the state of the game as seen by the client. The server keeps track of individual entities and players at all times and will send snapshots to the client calculated from the client’s field of view and detailing what the client should see.

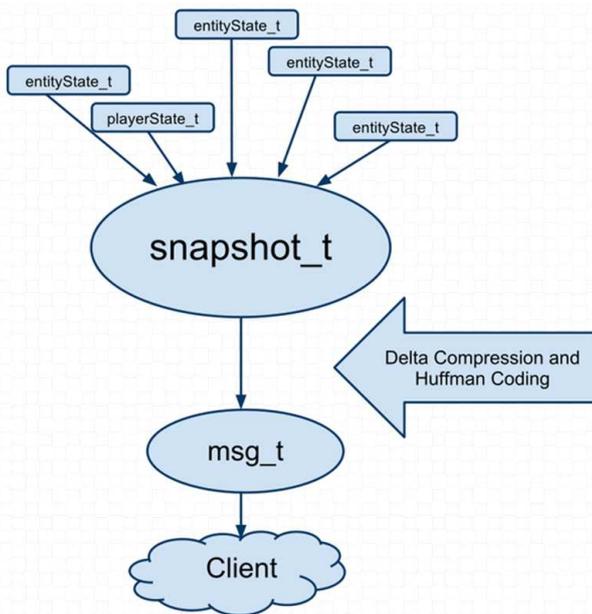


Figure 19. - Server to Client Communication

A. Snapshot Generation

To pass information back to a client, a *snapshot_t* is sent to provide the server's view at any particular point in time. Relevant entities (those that are in sight) and their *entityState_t* is sent together with an area mask, to indicate the applicable area covered. The *playerState_t* of the current player is often internally referenced as 'ps'. This provides the server's view of the current player's state. Due to use of local client prediction, dropped frames etc, server and client views of the local player may differ and may need to be overridden. To counteract cheating and inconsistent game states, the server's view maintains authority over the game.

It is probably worthwhile at this point to include the actual mechanism by which a client snapshot is generated. The *SV_BuildClientSnapshot* function determines which other entities are visible to the client, and copies the relevant player state and area mask bits for transmission.

A relevant segment of condensed code is included (Figure 20 on page 11) for illustrating this process.

Looking over this code, some reference points can be found for extrapolation. *ps* is the current player state based on the *playerState_t* struct which has been detailed earlier. *entityState_t* is the state of other entities which include bots, other players, power-ups etc. The current player state is extracted and other entities and portals within view are combined with the player's view using a logical OR operation. As mentioned earlier, a portal is

```

//// File: /code/server/sv_snapshot.c
// SV_BuildClientSnapshot
// Decides which entities are going to be visible to the client, and
// copies off the playerstate and areabits.
//
// For viewing through other player's eyes, client can be something
// other than
// client->gentity
static void SV_BuildClientSnapshot( client_t *client ) {
    clientSnapshot_t *frame;
    sharedEntity_t *ent;
    entityState_t *state;
    svEntity_t *svEnt;
    sharedEntity_t *client;
    int clientNum;
    playerState_t *ps;
// grab the current playerState_t
    ps = SV_GameClientNum( client - sv.clients );
    frame->ps = *ps;
// never send client's own entity, because it can
// be regenerated from the playerstate
// find the client's viewpoint
    VectorCopy( ps->origin, org );
    org[2] += ps->viewheight;
// add all the entities directly visible to the eye, which
// may include portal entities that merge other viewpoints
    SV_AddEntitiesVisibleFromPoint( org, frame, &entityNumbers,
    qfalse );
// now that all viewpoint's areabits have been OR'd together, invert
// all of them to make it a mask vector, which is what the renderer
// wants
    for ( i = 0 ; i < MAX_MAP_AREA_BYTES/4 ; i++ ) {
        ((int *)frame->areabits)[i] = ((int *)frame->areabits)[i] ^
        -1;
    }
// copy the entity states out
  
```

Figure 20. - Method for Snapshot Generation

not a teleport point, but is an element used to generate a map area and more information see Section II-E. An area mask is generated from the given information which will be included in the transmission and applied by the renderer to replicate the visible section of the map for the player. The method of generating a server snapshot can now be summarised.

- The current player's field of view is calculated
- Visible entities and portals are added
- All visible elements are combined to form a complete visible player state
- The snapshot state is saved for transmission
- The client game system presents this information to the renderer for display

Of further interest is a comment section at the start of the code block shown in Figure 20. An option is described for allowing a client to view the game through the eyes of another player. This routine provides the functionality for a player to view the game in a spectator mode, where they can follow other players.

B. Snapshot Processing

Delta updates are used to communicate changes in game state. Deltas are created from the differences in game snapshots. As such, it can be useful to look at how a snapshot is processed and sent. The process is similar on both the client and server sides, the difference being

```

//// File: code/client/client.h
// Game Snapshots are generated here:
// snapshots are a view of the server at a given
time
typedef struct {
    qboolean valid; // cleared if delta parsing was
invalid
    int messageNum; // copied from
netchan->incoming_sequence
    int deltaNum; // messageNum the delta is from
    byte areamask[MAX_MAP_AREA_BYTES]; // portalarea
visibility bits
    int cmdNum; // the next cmdNum the server
is expecting
    playerState_t ps; // complete information about
the current player at this time
    int numEntities; // all of the entities that
need to be presented
    int parseEntitiesNum; // at the time of this
snapshot
    int serverCommandNum; // execute all commands up
to this before
current
} clSnapshot_t;

```

Figure 21. - Client Side Snapshot Generation

```

//// File: /code/client/client.h
typedef struct {
    int timeoutcount; // it requires several frames
in a timeout condition
    clSnapshot_t snap; // latest received from
server
    gameState_t gameState; // configstrings
    int parseEntitiesNum; // index (not anded off)
into cl_parse_entities[]
    // cgame communicates a few values to the client
system
    int cgameUserCmdValue; // current weapon to add
to usercmd_t
    float cgameSensitivity;
    // cmds[cmdNumber] is the predicted command,
[cmdNumber-1] is the last // properly generated
command
    usercmd_t cmds[CMD_BACKUP]; // each message will
send several old cmds
    // big stuff at end of structure so most offsets are
15 bits or less
    clSnapshot_t snapshots[PACKET_BACKUP];
    entityState_t entityBaselines[MAX_GENTITIES]; //
for delta compression when not in previous frame
    entityState_t parseEntities[MAX_PARSE_ENTITIES];
} clientActive_t;

```

Figure 23. - Snapshot Buffer

```

//// File: code/client/cl_clgame.c
CL_GetSnapshot ()
(...)
    snapshot->serverCommandSequence =
clSnap->serverCommandNum;
    snapshot->ps = clSnap->ps;
    snapshot->entities[i] =
        cl.parseEntities[ (
clSnap->parseEntitiesNum + i ) &
(MAX_PARSE_ENTITIES-1) ];

```

Figure 22. - Client Side Snapshot Reconstruction

that the server keeps track of the entire game rather than just the entities in the local area. The components of a snapshot as seen by the client are specified in the client header file Figure 21.

Looking at this data structure, several components are of interest:

- areamask*: Describes the area visible to the player
- ps*: The complete state of the current player, based on *playerState_t*
- numEntities*: The number of entities requiring processing

The process of reconstructing a received snapshot on the client side is handled by *cl_clgame.c*, shown in Figure 22. This code fragment shows part of the reconstruction method used by the client to process a snapshot. The key steps are as follows:

- Snapshot is retrieved
- Server commands are processed
- Player state is processed
- Entities are enumerated and processed in turn

This effectively summarises the information required by the client for synchronization with the server state.

C. Storage of Snapshots

Although not essential for inclusion in this report, it is worth noting that a buffer of snapshots is retained for calculating delta states and also for local client prediction (Figure 23 on page 12).

The maximum amount of snapshots and entity states is governed through pre-defined variables in a central header file. The default code allows for up to 32 client snapshots to be stored.

VIII. SUB-TYPES OF A GAME ENTITY

The *entityType_t* structure was examined in section V-A4, this structure holds a large number of variables for any entity. This structure contains a sub-structure inside of it, *entityType_t*, or *eType*. To illustrate the sometimes recursive nature of how an entity manifests itself from within the game code, several entity subtypes are selected for further analysis. *entityType_t* acts as the master structure for detailing further subtypes. Every actionable item in the game is a type of entity. A player entity contains slightly more information than others, but for the sake of simplicity a weapon entity will be examined. For example consider a weapon entity that is on the ground awaiting pickup by a player, this case will be examined further in the following sections.

```

//// File: /code/game/bg_public.h
// entityType_t->eType
//
typedef enum {
    ET_GENERAL,
    ET_PLAYER,
    ET_ITEM,
    ET_MISSILE,
    ET_MOVER,
    ET_BEAM,
    ET_PORTAL,
    (...)
} entityType_t;

```

Figure 24. - Definition of Individual Entity Types through entityType_t (eType)

```

//// File: /code/game/bg_public.h
typedef struct gitem_s {
    char *classname; // spawning name
    char *icon;
    int quantity; // for ammo how much, or duration
of powerup
    itemType_t giType; // IT_* flags
} gitem_t;

```

Figure 25. - Structure for abstracting an item, gitem_t

A. entityType_t

Looking into this enum provides a list of different entity types defined within. A few of these are listed in Figure 24.

entityType_t can be seen to denote any actionable item within the game. Players are defined, items (ie. weapons, ammo and armor), missiles (ie. bullets and rockets) and movers (ie. doors and buttons). Further definitions are present, but have been removed for brevity.

B. gitem_t

The entities are further broken down further in individual structs as shown in Figure 25. Skipping ahead, an abstraction for general item types may be examined. It can be seen that *gitem_t* references another enum, *itemType_t*, which further expands on these properties. Using the method of recursion, more information can be obtained for the next structure.

C. itemType_t

Items are now seen to be defined further. Each item type can be defined with an individual set of properties, actions and behaviours as illustrated in Figure 26.

Further properties and behaviours of the entity are then forked out into separate files. For example, item behaviours and attributes such as respawning, pickups etc are contained within *g_items.c* (Figure 27).

```

//// File: /code/game/bg_public.h
// gitem_t->type
typedef enum {
    IT_BAD,
    IT_WEAPON, // EFX: rotate + upscale + minlight
    IT_AMMO, // EFX: rotate
    IT_ARMOR, // EFX: rotate + minlight
    IT_HEALTH, // EFX: static external sphere +
rotating internal
    IT_POWERUP, // instant on, timer based
} itemType_t;

```

Figure 26. - Definition of Individual Item Types through itemType_t

```

//// File: /code/game/g_items.c
// call the item-specific pickup function
switch( ent->item->giType ) {
    case IT_WEAPON:
        respawn = Pickup_Weapon(ent, other);
// predict = qfalse;
        break;

```

Figure 27. - Function for Dealing with an Encountered Weapon Object

It can be seen that *Pickup_Weapon()* is then called to deal with the case of *IT_WEAPON* being encountered. This is displayed in Figure 28 on page 13.

The function returns an integer indicating the delay until the weapon next respawns, *g_weaponRespawn.integer*. Two game entities are taken as arguments, the weapon entity (*ent*) and a player entity (*other*). Further sections of code have been omitted that code details actions taken during weapon pickup, such as adding the weapon to the inventory as well as the ammo contained within it. Although it may seem logical that an item is divided into sub-types, it can be useful to show just how prevalent they are throughout the Quake III Arena code.

The entity can be seen as a set of sub-components,

```

//// File: /game/g_items.c
int Pickup_Weapon (gentity_t *ent, gentity_t *other)
{
    int quantity;
    if ( ent->count < 0 ) {
        quantity = 0; // None for you, sir!
    } else {
        if ( ent->count ) {
            quantity = ent->count;
        } else {
            quantity = ent->item->quantity;
        }
    }
// add the weapon
other->client->ps.stats[STAT_WEAPONS] |= ( 1 <<
ent->item->giTag );
Add_Ammo( other, ent->item->giTag, quantity );
return g_weaponRespawn.integer;
}

```

Figure 28. - Code block for handling weapon pickups

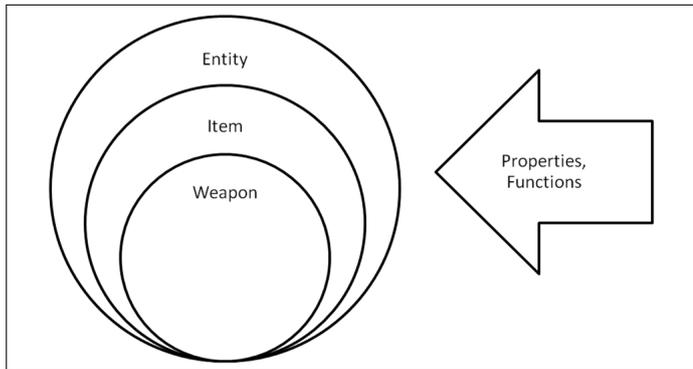


Figure 29. - Code block for handling weapon pickups

each of which can be referenced through individual functions for dealing with that level of the structure. Functions aimed at an entity will be more global in scope, perhaps to deal with snapshot generation, whereas functions aimed at a weapon would more likely deal with actions such as the amount of ammo contained within or the time before respawning after it is picked up by a player.

IX. CONSOLE COMMANDS

A. Gameplay Flags Affecting Packet Size

Commands may be entered through the console include those able to affect the size of network packets or to allow a better adjustment of network performance settings. These are of particular interest to anyone wishing to expand on points within this report.

```
/sv_nopredict 0 disable client-side prediction
/showpackets 0 enable network packet info display
/cl_nodelta 1 disable delta compression (default is 0)
```

By disabling delta compression, game states can be viewed directly within packets captured over the network. As the Huffman coding is disabled with this mechanism, network performance is obviously not optimised, however, this flag can be useful for reverse engineering packet structures and testing hypothesised changes to game code.

Certain opportunities also exist for receiving more than the normal amount information of information on other players from the server. When playing in team modes, a console command may be issued for allowing the server to send data on player states of other players. This can be initiated by using the console command: */cg_drawTeamOverlay 1* overlay states of other players onto map. Default is 0 (disabled)

In these cases, the *playerTeamState_t* struct is then used for conveying this information.

X. CONCLUSION

Quake III Arena is based on the “id Tech 3 Game Engine” and has been used by many other game titles since publication. Since the source code was made freely available, many modifications have been made to Quake III Arena, including complete remakes such as Open Arena.

Through the open nature of the Quake III Arena source code, an analysis of composition of structures for network communication was possible. This information can be used as a known constant when analysing other games of the FPS genre, where the source code may not be available.

The use of delta states was found to be a key identifier of the game. A baseline is established upon game initiation and client updates are sent at regular intervals to the server. The server maintains a global state of all entities within the game and periodically sends game snapshots to the client which describe the game view that the player should see. Game snapshots are sent as delta states which are based on changes to the game state since the last update. Delta states significantly reduce the size of the data needing to be sent as only a subset of information is required and Huffman coding is used to further reduce the size of these updates. When playing over lossy connections, the size of delta states may build up as packets are lost and remain unacknowledged, requiring them to be resent. Delta states can therefore be used to minimise network traffic, provided the connection loss is not too high. The server to client packets contain game state information that is based on the aggregation of the changes in state of the individual game entities in the vicinity of the player, whilst the client to server packets contain information of the changes in the individual player’s state.

XI. GLOSSARY

AAS:	Area Awareness System used to assist bots in navigating a map.
Bot:	Computer player used in Quake III Arena and most FPS games.
BSP:	Binary Space Partitioning, a method for recursive subdivision of spaces into planes, for improving rendering efficiency.
Enum:	An enumerated type is a data type used in programming, consisting of custom declared data types that are usually specific just to the defined enum.
FOV:	Field of View, the angle of the visual field (in degrees). 180 degrees is the standard

field of vision for humans.

Focus: Area within a field of view which defines the visible information a player can see.

FPS: First Person Shooter.

HUD: Heads up Display.

Leaf: Convex rendering section for representing a space between two portals. It may overlap with other leaves as a principle of design.

Mod: A set of modifications to the game commonly to change the look and feel of the game, such as through custom characters, different maps and so forth.

Nimbus: Area within a field of view which defines the visible information about a player.

Portals: Viewable areas between surfaces. These limit the areas that need to be rendered at one time. Segmented for small size viewpoints / interactions.

PVS: Potentially Viewable Set.

Struct: A C struct is a data structure that aggregates a number of data types into a single unit. The data types may be items such as integers or enums, and are not required to be of the same type.

REFERENCES

- [1] "id Software: Quake III," [Online], <http://www.idsoftware.com/games/quake/quake3-arena/>, accessed 7/10/2010.
- [2] "Quake 3 and Quake 3 Team Arena," [Online], available: <http://www.computerhope.com/games/games/q3.htm>, accessed 26/11/2010.
- [3] "Open Arena," [Online], available: <http://openarena.ws/smfnews.php>, accessed 10/11/2010.
- [4] "Quake 3 Source Code," [Online], available: <http://www.idsoftware.com/>, accessed 26/11/2010.
- [5] T. Lang, P. Branch, and G. Armitage, "A synthetic traffic model for Quake3," in *ACE '04: Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*. New York, NY, USA: ACM, 2004, pp. 233–238.
- [6] H. Park, T. Kim, and S. Kim, "Network traffic analysis and modeling for games," in *Internet and Network Economics*, ser. Lecture Notes in Computer Science, X. Deng and Y. Ye, Eds. Springer Berlin / Heidelberg, 2005, vol. 3828, pp. 1056–1065.
- [7] Z. S. Bussiere J.A., "Enemy Territory traffic analysis," Swinburne University of Technology, Melbourne, Australia, Tech. Rep., 2006.
- [8] G. Armitage, "An experimental estimation of latency sensitivity in multiplayer quake 3," in *11th IEEE International Conference on Networks (ICON 2003)*, Sydney, Australia, 28-1 September 2003, pp. 137–141. [Online]. Available: <http://dx.doi.org/10.1109/ICON.2003.1266180>
- [9] "id tech 3 (video game concept)," [Online], available: <http://www.moddb.com/engines/id-tech-3>, accessed 26/11/2010.
- [10] "Id tech," [Online], http://en.wikipedia.org/wiki/Id_Tech, accessed 26/11/2010.
- [11] "ioquake3," [Online], Available: <http://ioquake3.org/>, accessed 7/10/2010.
- [12] "id tech 3," [Online], available: http://en.wikipedia.org/wiki/Id_Tech_3, accessed 7/10/2010.
- [13] Tei, "File:quake - family tree.svg," [Online], August 2010, available: http://en.wikipedia.org/wiki/File:Quake_-_family_tree.svg, accessed 10/11/2010.
- [14] G. Armitage, M. Claypool, and P. Branch, *Networking and Online Games - Understanding and Engineering Multiplayer Internet Games*. UK: John Wiley & Sons, 2006.
- [15] D. Tabor, "The guide - first-person shooters," [Online], available: <http://www.bluesnews.com/guide/fps.htm>, accessed 10/11/2010.
- [16] M. H. Brian Healy, "A beginner's guide to pc gaming: First person shooters (fps games) - half-life, crysis, far cry & more," [Online], <http://www.brighthub.com/video-games/pc/articles/840.aspx>, accessed 10/11/2010.
- [17] "Quake 3 excellent award screenshot," [Online], available: http://www.freakygaming.com/gallery/action_games/quake_3/excellent_award, accessed 26/11/2010.
- [18] J. P. Costella, "A Beginner's Guide to the Human Field of View," [Online], November 1995, http://www.assassinationscience.com/johncostella/physics/A_beginners_guide_to_the_human_field_of_view.pdf, accessed 26/11/2010.
- [19] G. Alexander, "Space odysseys: Glossary," [Online], <http://archive.artgallery.nsw.gov.au/sub/spaceodysseys/glossary.html>, accessed 18/11/2010.
- [20] H. H. Jouni Smed, Timo Kaukoranta, "A review on networking and multiplayer computer games," Turku Centre for Computer Science, Tech. Rep. 454, April 2002. [Online]. Available: <http://staff.cs.utu.fi/~jounsm/papers/TR454.pdf>
- [21] "Tweaking Q3 Arena Engine," [Online], January 2004, available: <http://kpush.tripod.com/tweaking/id4.html>, accessed 10/11/2010.
- [22] A. Champandard, "Analyzing the AI bot library from the Quake 3 source code," [Online], January 2008, <http://aigamedev.com/open/highlights/quake3-engine/>, accessed 4/11/2010.
- [23] Mr Elusive, "GtkRadiant Editor Manual: Appendix C," [Online], January 2000, https://zerowing.idsoftware.com/svn/radiant.gamepacks/Q3Rad_Manual/trunk/appndx/appn_c.htm, accessed 17/11/2010.
- [24] J. van Waveren, "The Quake III Arena Bot (1st Rev.)," [Online], June 2001, available: <http://dev.johnstevenson.co.uk/bots/20585341-The-Quake-III-Arena-Bot.pdf>, accessed 17/11/2010.
- [25] "Binary space partitioning," [Online], October 2010, http://en.wikipedia.org/wiki/Binary_space_partitioning, accessed 4/11/2010.
- [26] "Quark glossary," [Online], July 2009, available: <http://quark.sourceforge.net/infobase/glossary.html>, accessed 25/11/2010.