

Faculty of Science Summer Scholarship Report

Information Visualisation utilising 3D Computer Game Engines

Case Study: A source code comprehension tool

Blazej Kot

Supervised by:

Burkhard Wuensche

John Hosking

John Grundy

February 2005

Department of Computer Science,
University of Auckland

Table of Contents

1	ABSTRACT	3
2	INTRODUCTION	4
2.1	MOTIVATION	4
2.2	OUTLINE OF REPORT STRUCTURE	4
3	BACKGROUND	5
3.1	COMPUTER GAMES AND GAME ENGINES	5
3.1.1	<i>Game Genres</i>	5
3.1.2	<i>Game Architecture</i>	6
3.1.3	<i>Available Game Engines</i>	7
3.1.4	<i>Quake 3</i>	9
3.2	SPATIAL MEMORY AND 3D USER INTERFACES.....	11
3.3	SOFTWARE VISUALISATION.....	12
3.4	PREVIOUS USES OF GAME ENGINES FOR VISUALISATION	15
4	INFORMATION VISUALISATION USING A GAME ENGINE	17
5	DESIGN OF THE SOURCE CODE COMPREHENSION TOOL	19
5.1	REQUIREMENTS FOR THE TOOL	19
5.2	REQUIREMENTS FOR THE GAME ENGINE AND ENGINE CHOICE	20
5.3	THE USER INTERFACE AND INTERACTION METAPHOR (SINGLE USER).....	21
5.4	MULTI-USER FEATURES	23
5.5	DETAILED DESIGN AND HIGH-LEVEL OVERVIEW OF IMPLEMENTATION	25
5.5.1	<i>Source Code Analysis and Cross-Referencing</i>	25
5.5.2	<i>Quake 3 Modification</i>	26
6	TOOL IMPLEMENTATION	28
6.1	THE DOX2HTML TOOL.....	28
6.2	THE QUAKE 3 MODIFICATION	29
6.2.1	<i>Server-side Modifications (game.qvm)</i>	29
6.2.2	<i>Client-side Modifications (cgame.qvm)</i>	37
6.2.3	<i>Modifications to the UI (ui.qvm)</i>	40
7	TOOL USAGE	43
7.1	INSTALLATION.....	43
7.2	EXAMPLE OF MULTI-USER CODE WALKTHROUGH	44
7.3	SETTING UP THE TOOL TO VISUALISE A DIFFERENT SOURCE PROJECT	51
8	EVALUATION	53
8.1	EVALUATION OF THE SOURCE CODE COMPREHENSION TOOL	53
8.2	EVALUATION OF QUAKE 3 GAME ENGINE SUITABILITY	54
9	CONCLUSION	56
10	FUTURE WORK	57
11	REFERENCES	58
12	ACKNOWLEDGEMENTS:	59

1 Abstract

The overall aim of this project was to investigate how technology developed for computer games may be reused for creating collaborative information visualisation tools. The investigation took the form of implementing a source code comprehension tool using the Quake 3 game engine. Based on this experience, it was found that game engines can be a good basis for an information visualisation tool, provided that the visualisations and interactions required meet certain criteria, mainly that the visualisation can be represented in terms of a limited number of discrete, interactive, and physical entities placed in a static 3 dimensional world of limited size.

The tool which was developed presents individual source code files of a project as entities which can be moved around in a physical 3D environment. Furthermore, the source code is displayed with hyperlinks, so that it is easy to jump to the definition of any symbol in the project. Full collaborative support exists, whereby multiple users can chat and point out parts of source code to each other. Users are able to “lock” their view with any other user’s, so that one user is able to give a “guided tour” of the source code to other users.

2 Introduction

2.1 Motivation

Modern computer games make use of technologies from almost all areas of computer science: graphics, artificial intelligence, network programming, operating systems, languages and algorithms. A modern computer game, such as Doom 3 or Unreal Tournament 2004 contains efficient, well-tested implementations of hundreds of computer science ideas. It therefore makes sense to investigate possible ways of reusing these implementations for various tasks, thus saving development time.

Computer game implementations have been previously applied to some alternative tasks, such as visualising architectural designs[1], military simulation visualisations[2], landscape planning visualisation[3] and as an interface for Unix process management[4].

This project focuses on utilising computer game implementations for information visualisation. This seems to be a relatively unexplored area, as we could only find a few instances of work related to this topic[4,5].

In particular, this project investigates implementing a source code comprehension tool. This example was chosen since code comprehension is about the user exploring a structure, and remembering what is where. It was hoped that this would map well onto a 3D environment, where the spatial memory of the user could be engaged to remember the layout of the code structure.

2.2 Outline of Report Structure

Section 3 presents the background, including a discussion of game engines, the role of spatial memory in 3D user interfaces, previous work related to 3D software visualisation and selected existing alternative uses of game engines. Section 4 briefly discusses how game engines can be used for information visualisation. Section 5 and 6 present the design and implementation of the source code comprehension tool. Section 7 provides a demonstration of usage of the tool. Section 8 evaluates the tool and the suitability of using game engines for information visualisations. Section 9 is the Conclusion, followed by Future Work, Section 10, and Acknowledgements, in Section 11.

3 Background

3.1 *Computer Games and Game Engines*

3.1.1 Game Genres

There are many computer game genres. Out of the genres which use graphics (as opposed to text based games) the main ones are First Person Shooter (FPS), Real Time Strategy (RTS) and Role Playing Game (RPG). All genres contain games which are either single player (wherein other players are simulated using artificial intelligence) or multi-player (where several players can interact in the same virtual world via a computer network), or both.

In a FPS game, the player travels around in a three dimensional world, shooting “bad guys.” In a RTS game, the player views a two dimensional map with many units (for example, of an army) on it. They control their own units and use them to attack and defeat their opponents. An RPG is similar, except the player controls only one unit, their “character”, via which they explore a 2D (e.g. Age of Empires II [6]) or 3D (e.g. World of Warcraft [7]) world.

Computer games can also be divided into those that run on PCs and those that run on consoles, which are purpose-made gaming computers usually connected to a TV. In this project, only PC games will be considered, since the hardware is much more ubiquitous, and the cost of development tools for console games is prohibitive.

In this project, only FPS game engines will be discussed, however future work may involve investigating other genres.

3.1.2 Game Architecture

Most modern computer games can be split into three parts: the game engine, the game logic and the game art. The game engine is the main executable file which runs on the computer. It provides an environment within which the game logic runs, as well as basic mathematics, graphics, audio, user input and network functions. The game logic may take the form of scripts, bytecode for a virtual machine, or a library (a DLL for example). The game logic's task is to control the game play, and to use the engine to display the game art as appropriate. The game art consists of things such as pictures (*textures* in game parlance), *maps* (layouts of virtual worlds), *models* (3D representations of things inhabiting the world, for example players, weapons or flowerpots) and sounds.

It is possible to use a game engine for creating many different computer games, by altering the game logic and art. Such alterations are called “modifications” or “mods” for short. Creating a mod is the easiest way of customising an existing game. There is an active community of amateur “modders” for every game, as well as the occasional commercial mod (for example, Return to Castle Wolfenstein is based on the Quake 3 engine). Extensive mods are referred to as “total conversions,” in which all the game logic and art has been replaced.

In some cases, it is possible to modify the game engine itself, if its source code is available. There are several open source game engines, discussed below, as well as older commercial engines which were open-sourced by their developer, such as Quake 2.

3.1.3 Available Game Engines

Game engines can be divided into two categories: open source and closed source.

3.1.3.1 Open Source Game Engines

Open source game engines are either ones written by amateurs, or older commercial engines which the developer decided to open source.

Out of the amateur ones, the main ones are:

OGRE[8]

A mature multi-platform engine, supporting OpenGL and DirectX rendering, written in C and C++, under the LGPL licence. It has an OO based design, and supports loadable code modules. It provides a hierarchical scenegraph, basic physics functions, and has a 2D GUI system built in, supporting common widgets and TrueType fonts. It has a large and active support community of developers and users. It is however mostly a graphics engine, with no support for networking or audio.

Crystal Space[9]

Another popular and well-supported engine based on OpenGL, which runs on Windows, Linux and MacOS, with a LGPL licence. Based on a OO design, with support for code plugins. Provides simple networking and audio functions as well as a 2D GUI system.

Irrlicht[10]

This engine can use either OpenGL, DirectX or software for rendering. It runs on Windows and Linux, and is under the zlib license. It has a reputation as having poorer graphics quality and rendering speed than the others, which is offset by being easier to develop with[12]. It is often viewed as the ideal beginner's engine[12]. It uses a hierarchical scenegraph for scene management, and has basic 2D GUI functionality. It is somewhat smaller than the engines above, with only one main developer, however there is a large support community of enthusiasts. It also provides basic physics routines, but lacks any network functions.

The Nebula Device 2[13]

This is a Windows only, DirectX based engine. It was not investigated further.

OpenSceneGraph[14] and OpenSG[15]

These are not game engines, rather purely graphical engines. However, they do provide an easy way of creating an interactive 3D world, so they are mentioned for completeness. They are based on OpenGL, and run on many platforms. They lack physics, audio, network and 2D GUI functionality.

In addition to these, there are the Doom, Doom 2, Quake and Quake 2 engines which have been open-sourced by id Software[16]. They run on Linux and Windows, are OpenGL based, and since they are from a real commercial game they include support for all the gaming features such as physics, audio, network, 2D GUI, etc. They suffer from being older, providing poorer rendering quality than the newer engines. Quake and Quake 2, the newest of the four, still have a sizeable amateur modding community.

3.1.3.2 Closed Source Game Engines

Currently, in the FPS genre, there are three main game engine families, each from a different developer: Doom 3 and Quake 3 engines from id Software[16], Half Life and Half Life 2 engines from Valve Software[17] and Unreal Tournament (UT) and Unreal Tournament 2004 (UT2004) engines by Epic Games[18]. Doom 3, Half Life 2 and UT2004 represent the latest generation from each developer, and are the best game engines available. Quake 3, UT and Half Life are the previous generation engines. It should be noted that the Half Life engine is based on the Quake and Quake 2 engines.

All of these engines are fully featured, and there exist one or more complete games based on each of these engines. This is in contrast to most of the open source engines above, which provide more basic functionality. The open source engines often need to be combined with other libraries and toolkits to create a playable game, while the six engines listed here have all of the required functions built in. Out of these six, Quake 3 deserves special mention as id Software plans to open source it in the near future[19]. This would mean that, unlike the other engines here, extensive modifications to the game engine would be possible. (Mods for the other ones are restricted to altering the game logic and art.)

Another engine which should be mentioned is Torque by GarageGames[11]. It is an inexpensive commercial engine. It was not considered for this project.

3.1.4 Quake 3

As the Quake 3 engine was chosen for use in this project, this section gives a very brief overview of the engine and its architecture. For more information see [20,21], although the documentation available is rather limited.

Quake 3 uses the standard FPS game control system: mouse look and keyboard. Moving the mouse around changes the direction the player looks in. The mouse buttons are typically used for walking forwards and for shooting. Various keys on the keyboard are used for crouching, jumping, moving backwards, strafing and switching weapons. The keys can be remapped to different in-game functions via a process known as *key binding*.

Quake 3 is by design a network-oriented (LAN or internet) game, using the client-server model of communications. Each computer running Quake 3 runs an instance of `quake3.exe`, the game engine. This executable is capable of running bytecode for three virtual machines: game, cgame and UI, stored in the files `game.qvm`, `cgame.qvm` and `ui.qvm`. “QVM” refers to Quake Virtual Machine. `game.qvm` is the server part of the game. It is responsible for maintaining the state of the game world, such as positions of all entities, and sending messages to the clients. It also has the final say on issues such as whether a certain bullet hit a certain player or not. `game.qvm` does not do any rendering; it only communicates with clients. `cgame` is the client QVM – there is one running on each computer connected to a particular game. The client is responsible for rendering the map and entities, according to data sent by the server. Additionally, the client will interpolate the positions of entities in the world between snapshots sent by the server, to minimise the effect of slow network connections or dropped packets. Finally, `ui.qvm` is responsible for displaying the in-game menus. It is the first QVM started, and it will start the game and/or `cgame` QVMs when the user, via the menu system, chooses to start a new game.

The QVMs are all coded in ANSI C. They are then compiled by a compiler provided by id Software, which outputs bytecode that can be run by the `quake3.exe` engine. The virtual machine does not support dynamic memory allocation (no `malloc()`).

Within the QVM environment there are several available *system calls* or *traps*. These are functions that can be called within the code of the QVM, to pass control into the main `quake3.exe` executable. This is how tasks such as drawing on the screen, file and network

access are carried out. All these functions have `trap_` prefixes, e.g. `trap_FS_FOpenFile()`.

Note that individual polygons are not drawn via system calls – the `trap_` functions deal with higher-level data such as models or maps. This is of course to improve performance, since drawing all individual polygons from within an interpreted VM would introduce unnecessary overhead.

In a normal multi-player game, one computer is the server. This computer will be running `quake3.exe`, which will in turn be running `game.qvm` (for the main game server logic) and `cgame.qvm` (to provide a client on that computer, so a player may use it to play in the game). Other computers, clients, may join the server; these computers will be running `quake3.exe`, which will be only running `cgame.qvm`. It is also possible to run a dedicated server, that is having `quake3.exe` only run `game.qvm`, for increased server performance.

The game data such as textures, audio and models is kept in `.pak` files. These are just a zip archive of all the individual files. There is a special file hierarchy that needs to be used within the zip files, such as all sounds are placed in `sounds\`, the QVMs in `qvm\` and so on. For development purposes, files may also be placed outside the `.pak`, but following the same file hierarchy as if they were in it. If then the in-game option `sv_pure` is set to 0, these files can be accessed by the game code normally.

Quake 3 provides a drop-down console, which can be activated at any point in the game or in the menus by pressing the tilde `'~'` key. There, commands can be typed, such as `\connect 192.168.0.12` to connect to a game server, or variables can be set, such as `\sv_pure 0`. Variables that can be altered via the console are called *cvars*.

Internally, the main message passing mechanism (or rather, the most easily accessible and modifiable one) is based on passing variable-length, null-terminated strings. These can be sent from the server to the client (a *ServerCommand*) or vice-versa (a *ClientCommand*). Additionally, the client can send messages to the UI QVM via `trap_SendConsoleCommand()`. The UI can also send messages directly to the server, via the `trap_Cmd_ExecuteText()` call, although in this case the command is only appended to the cue of commands waiting at the server, not immediately executed.

Currently, only the Quake 3 game logic code is open to the public. This consists of a total of approximately 100 000 lines of ANSI C code running on the three Quake 3 virtual machines.

3.2 Spatial Memory and 3D User Interfaces

Spatial memory refers to the ability of humans to remember where things are placed in the physical world. Some people have better spatial memory ability than others, and several papers[22,23,24] provide evidence that people with better spatial memories will perform better when using certain computer user interfaces.

Some research has been done regarding whether spatial memory is best utilised in 2D or 3D interfaces. The results of Cockburn et al.[25] seem to indicate that users find a 3D user interface more confusing and cluttered, hence eliminating any possible benefits due to improved spatial memory use. In this paper, an experiment was carried out by having subjects recall the location of items in a 2D, 2.5D and 3D spaces, both physical and virtual (on a computer screen). The main result was that the subjects performed best with the 2D interface.

The paper above also mentions that adding semantic labels to elements in a 3D space may improve user performance to above that of a standard 2D interface. The tool implemented in the present project places the elements in virtual rooms and hallways in a 3D world, unlike the experiment by Cockburn et al. where the items were all floating in space, thereby hopefully providing a type of semantic label, and so overcoming the confusion that may arise from a plain 3D interface.

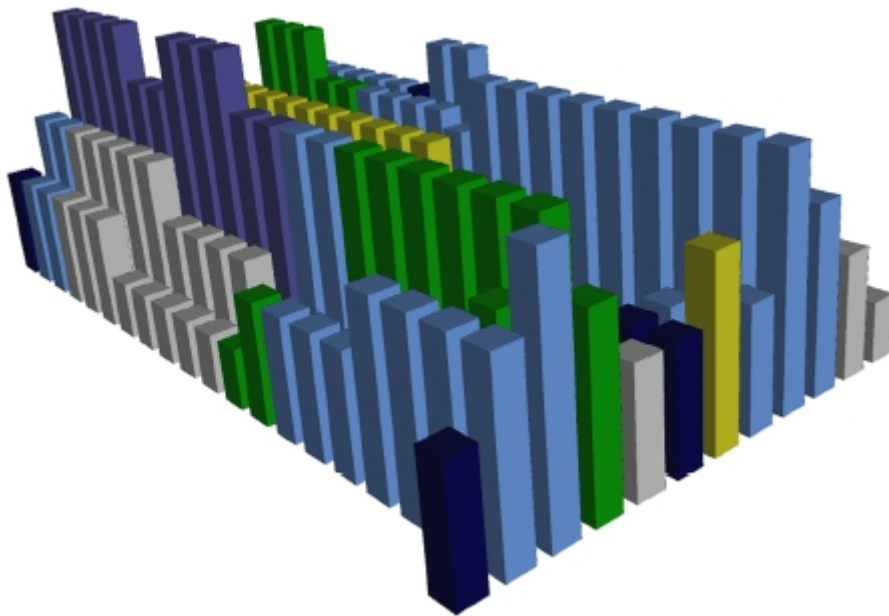
In a paper on 3D software visualisation[26] there is a very good summary of research about the advantages 3D visualisation, which is quoted below, due to its applicability:

“The work of Hubona, Shirah and Fout [Hubona et al. 1997] suggest that users' understanding of a 3D structure improves when they can manipulate the structure. Ware and Franck [Ware and Franck 1994] indicate that displaying data in three dimensions instead of two can make it easier for users to understand the data. In addition, the error rate in identifying routes in 3D graphs is much smaller than 2D [Ware et al. 1993]. The CyberNet system[Dos Santos et al. 2000] shows that mapping large amount of (dynamic) information to 3D representation is beneficial, regardless of the type of metaphors (real or virtual) used. Also, 3D representations have been shown to better support spatial memory tasks than 2D [Tavanti and Lind 2001]. In addition, the use of 3D representations of software in new mediums, such as virtual reality environments, are starting to be explored [Knight and Munro 1999, Maletic et al. 2001].”[26]

3.3 Software Visualisation

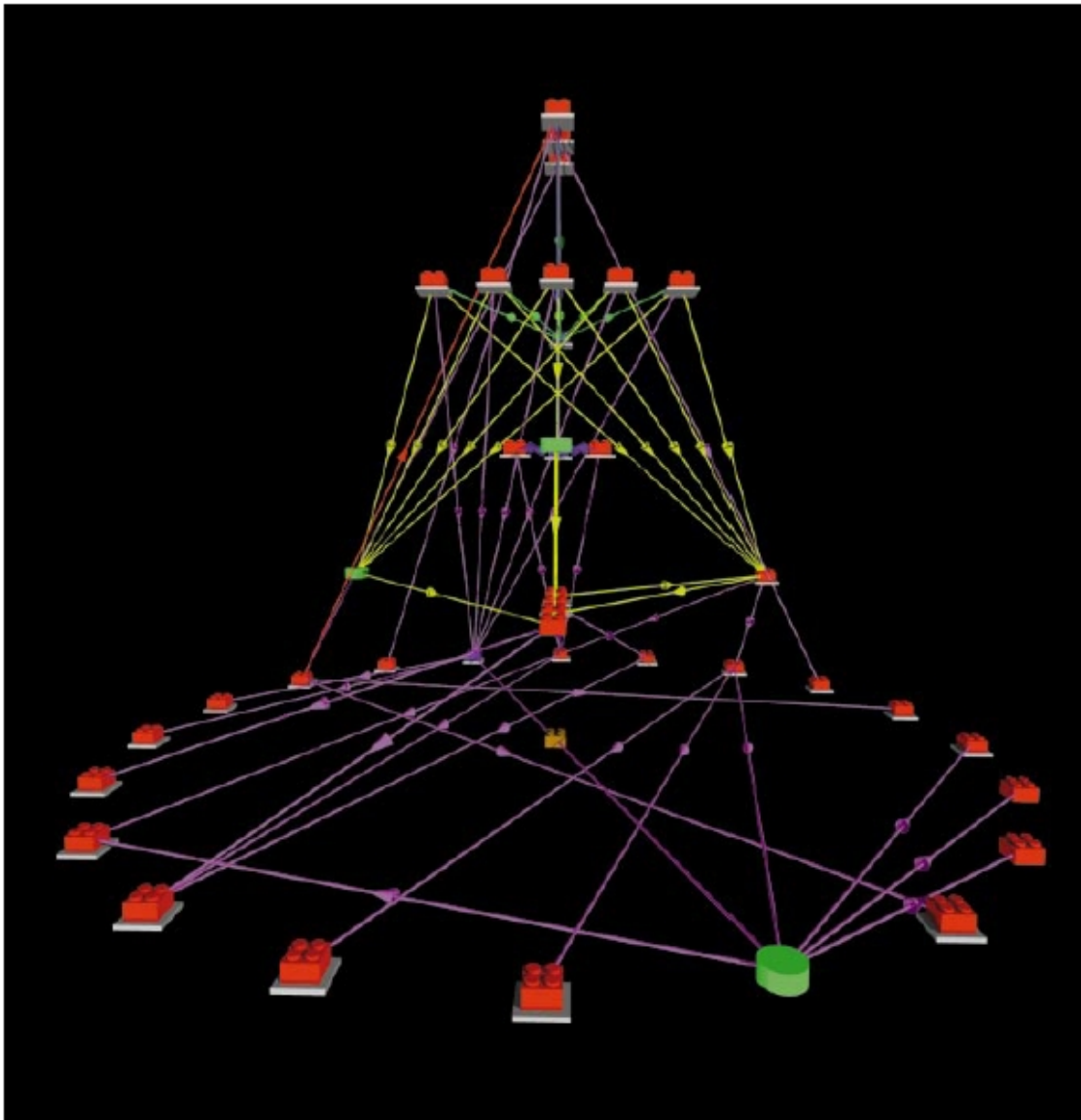
A substantial amount of work has been done in the area of software visualisation. In this section, only several selected examples of 3D software visualisation are presented.

SeeSoft[27], is one of the better-developed 3D software visualisations. A sample visualisation using the sv3D[26] implementation of this system is shown below, taken from [26]:



This figure shows one source file, each line of which is represented by a cuboid. In this example, the colour of the cuboid represents the control structure type, while the height represents the nesting level of the line. The sv3D implementation allows the user to arbitrarily map other attributes of the code lines to various graphical attributes, such as transparency.

Another interesting 3D visualisation is ArchView[28]. Here, individual source code modules are represented by “LEGO blocks” floating in a 3D space. Import relationships between modules are shown by arrows, while the type of module (a user-specified categorisation) is represented by the colour and shape of the bricks. Below is an example taken from the ArchView paper, showing the architecture of a medium-sized project.



Yet another interesting visualisation[29] uses a 3D city metaphor to represent a software project. In the below image taken from that paper, Java classes are represented by individual buildings, whose size represents the number of source code lines in that class. The spacing of buildings shows the amount of coupling between the classes, and the type of building indicates the quality of the code – old and collapsed buildings represent code which needs to be refactored. Cars travelling within this city show the dynamic execution path of the visualised program. Since the cars leave traces, places of heavy traffic can be identified, and these correspond to heavy communication between classes.

Various aspects related to software project management are then superimposed on top of this view, such as surrounding often executed classes by flames and coloring parts of the code which are not used brown.



3.4 Previous uses of Game Engines for Visualisation

PSDoom[4] is a utility for process management, implemented as a modification of the Doom computer game. It provides the functionality of the Unix `ps` command via a 3D user interface. Running processes are represented as monsters (enemies), which can be shot and killed, thereby killing the associated process. Monsters can fight back, and more important processes are represented by bigger monsters (which are more difficult to kill), thereby reducing the chance that they will be killed. Interestingly, when many processes are running, and the 3D space becomes crowded with monsters, the monsters start attacking each other (a normal Doom behaviour). This provides a natural control mechanism for processes in a heavily loaded system – less important monsters will be killed first, since the important monsters are represented by stronger monsters. The screenshot below shows the user killing the emacs process.



Heckenberg et al.[5] implement a visualisation of a simplified financial market, “The Minority Game,” using a modification of the Unreal Tournament 2003 computer game. The Minority Game consists of two teams of agents. These agents are represented as 3D players in the game. The Minority Game centres on the agents making decisions so as to end up on the “winning team” (which is defined as the team with least agents.) In the current implementation, this is all that the agents do – swap teams, which can be seen by viewing the scoreboard. Future work is suggested to make use of other game features, such as players being able to throw stocks and money at each other, to perform trading. The screenshot below shows the scoreboard where the agents can be observed as they swap teams.



Computer games can also be used to visualise less abstract concepts, such as in [2], where Unreal Tournament is used to visualise a battlefield in 3D in an army simulation. Another relevant paper[3] discusses a tool for landscape design and planning based on a game engine. Moloney et al.[1] have used a game engine for visualising architectural designs.

4 Information Visualisation using a Game Engine

There are two main ways in which a FPS game engine can be used for information visualisation. One way is to modify an existing game and only add the features necessary for the visualisation, leaving the basic style of interaction with the 3D world intact. The other way is to totally rewrite the game logic, and only make use of the graphics, audio and networking functionality provided by the engine itself. This approach is more flexible with regards to what visualisations can be created, however it requires a lot more work on the part of the developer. In fact, this approach is similar to using a visualisation toolkit or engine, such as OpenSG[15]. In this project, only the former approach is considered, as this is the option that allows maximal reuse of the computer game implementation.

In an FPS game, there are two primary types of elements: a static, or almost static, map (layout of rooms) and dynamic, interactive entities occupying positions in this map.

There are many different ways to represent parts of an information visualisation by game elements. The particular mapping chosen obviously depends on the particular visualisation. For example, in a visualisation of a file hierarchy, the layout of directories could be represented by the layout of the rooms (that is, the map), while files are entities occupying positions within these rooms.

One thing to note that may affect this mapping is that in most current FPS games (specifically, Quake 3) the map can not be altered during a game session. This could be somewhat worked around as players can be moved between maps relatively easily, so that one map could be altered while the players are in another map, creating the illusion of a dynamic world. This has the disadvantage that the game will pause while switching maps, which may rule out this work around for certain applications. Additionally, Quake 3 maps are limited in size, which may also be worked around by splitting a large map into several smaller maps, but with the same problem of the game pausing between map changes.

Another issue is that FPS games are designed for relatively low amounts of entities – Quake 3 only allows a maximum of 1024 entities in a map. With access to the game engine source code, this limitation may be worked around, but this may introduce a performance hit. The upshot of this is that in some cases it may make more sense to represent parts of the visualisation as dynamically generated textures (e.g. a diagram of a graph structure) rather than as separate entities (e.g. trying to represent each node in a large graph as a separate entity). Again, this could be worked around by using multiple maps, or by spending some time extending the engine.

Yet another peculiarity of game engines is that they are designed to only support one style of interaction, the one defined by the game logic. For example, in Quake 3 each entity usually has a fixed appearance, and a fixed behaviour throughout a game session. (It is actually possible to alter these programmatically during a game session in the game logic, if desired.) The problem is that the engine does not provide any “multiple view types” support. So, if the visualisation to be implemented relies on multiple view types (e.g. seeing files first as parts of a pie chart of disk usage, and then as entities inhabiting 3D rooms), one must be prepared to code a framework on top of the engine which will keep track of what view is being currently used, and tell the game logic which representations and behaviours to use for which entity.

5 Design of the Source Code Comprehension Tool

5.1 Requirements for the Tool

The main purpose of the tool is to illustrate how game engines may be used for information visualisation. Therefore, the tool should use a visualisation metaphor that maps simply and naturally into a static 3D world occupied by dynamic, interactive entities. The metaphor that was chosen is to represent source code files as entities in a static 3D world. They can be moved around at will, to arrange them in logical groupings. Files can be viewed by walking up to them. The source files are cross-references using hyperlinks, so that clicking on a symbol in the source code takes the user to a definition of that symbol. Also, back and forward buttons exist so that the user can walk through their history, like in a web browser.

The metaphor used by the tool is essentially just a 3D version of a web browser, using cross-referenced source files as the web pages. The advantages of using a game engine instead of a normal browser are mainly: multi-user support either via a LAN or the internet, so one user can guide another user via the code; and utilisation of the user's spatial memory, since closely-related files can be placed together in the 3d world – hopefully making it easier for the user to comprehend the structure of the source code.

Throughout the design of the tool, the use scenario considered the most was that of a new employee, or group of employees, arriving to work at a company, and being given a guided tour of the company's source code base by an existing employee.

The main functional requirements identified are:

- ◆ A web browser like interface, but with webpages (source code files) represented as movable 3D entities in a 3D world. Syntax highlighting and hyperlinks are to be displayed when a source code file is shown.

- ◆ Collaborative support:
 - Ability for users to identify, by their appearance (model), other users in the world.
 - Ability for a user to give a guided tour – so that the new employees can automatically follow the experienced employee.
 - Ability for users to point out specific parts of the code to each other during a guided tour – by circling parts of code on their screen, and having the same drawing appear on the other player’s screens.
 - Communication between users (chat)

5.2 Requirements for the Game Engine and Engine Choice

The game engine used needs to be multi-user capable, stable, and well tested. Additionally, since as discussed in Section 4 the tool will be implemented by modifying an existing game running on the chosen game engine, there must be a well-tested, open-source, implementation of a game for the chosen game engine.

The game engine which seems to best meet these requirements is the Quake 3 engine, with the corresponding game implementation, Quake 3 Arena. The Quake 3 engine source code is at the moment not available to the public. The Quake 3 Arena source code is available, under a certain licence (which does appear to permit modifying the source code and distributing the modified game virtual machine bytecode – see the licence for details.).

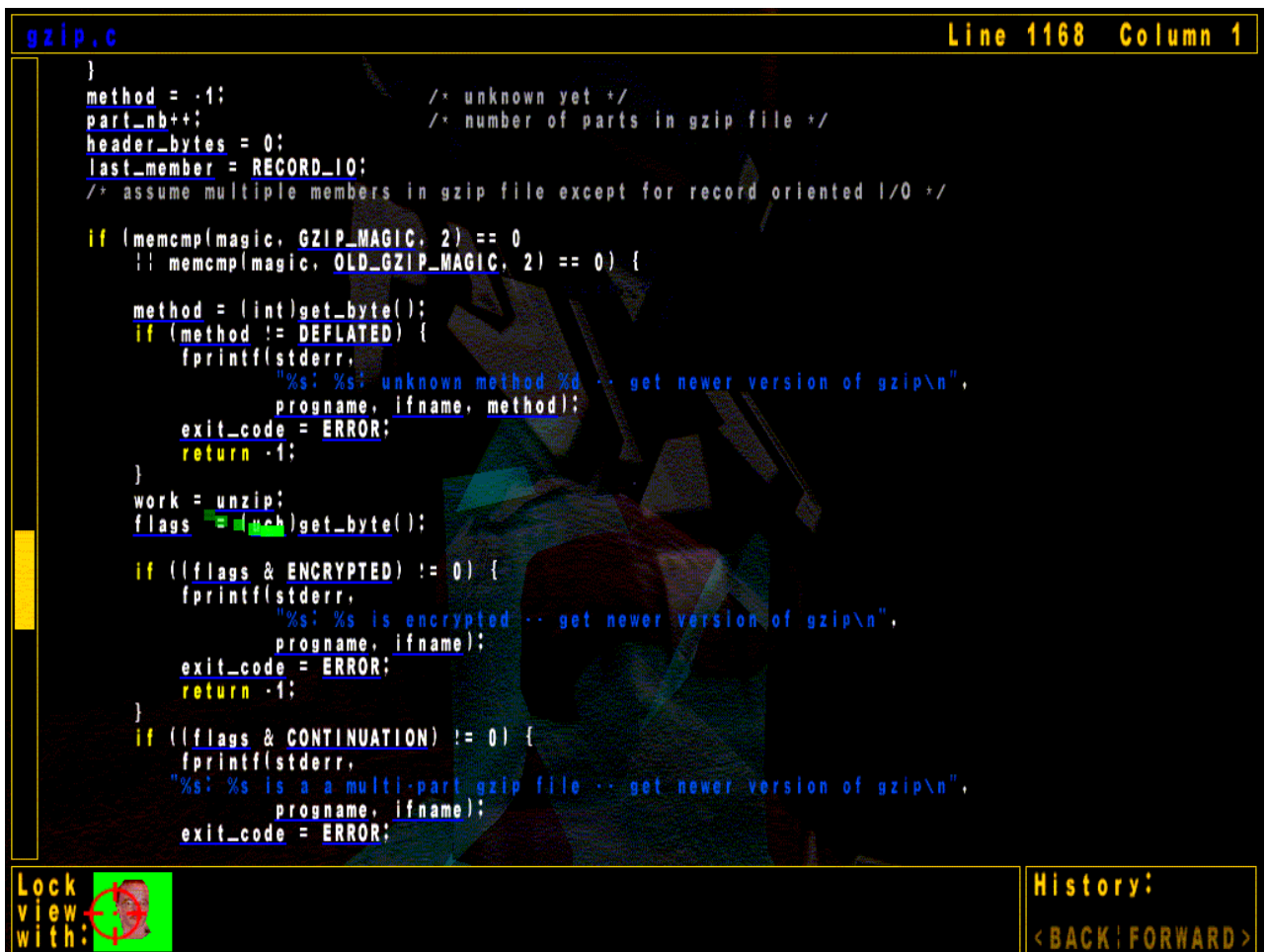
The open source game engines, such as Ogre, Crystal Space and Irrlicht (see section 3.1.3), were investigated, however most of them lacked crucial features (such as networking support), or had no well-tested game implemented using them.

5.3 The User Interface and Interaction Metaphor (Single User)

The tool is based on displaying source code files as individual entities within a 3D world. These entities can be moved around, much like icons can be moved around on a 2D desktop. The entities are drawn as floating “T” shapes, for no reason other than that this seemed to be the most appropriate of the existing entity models in Quake 3. (Another model could be easily substituted in the future.) The size of the drawn entity indicates the size of the corresponding source file. There is a minimum size (approximately the size of a player’s head), which is used for empty files, and a maximum size (slightly bigger than a whole player), which is used for files of 100 kilobytes and more. Between these two extremes the size relationship is linear. All file entities have their filename displayed above them. This is done in such a way as to have the text always be facing the player, and be of constant size, irrespective of the distance between the player and the file. This was done to improve the readability. Additionally, header files (detected as those files whose name ends in “.h”) have their filenames displayed in red, and their “T” shapes surrounded by a spheroid, while other files have their filenames in blue, and have no surrounding spheroid. Below are shown a small header file (left) a large .c file (right):



Files can be picked up by shooting at them. When this happens, the file disappears from the map, in all players' views, but a "T" icon appears on the side of the screen of the player who is now holding the file. The player can then walk anywhere on the map, and press the e key (or whatever key is bound to the item use quake function) to drop the file at the new location. The file then appears at this new place in the map, in all players' views. In the current implementation, there is no way for players to distinguish if a certain player is carrying a file. This may cause some confusion due to "missing" files, so this feature should be add in a later version.




```
gzip.c Line 1168 Column 1
}
method = .1; /* unknown yet */
part_nb++; /* number of parts in gzip file */
header_bytes = 0;
last_member = RECORD_IO;
/* assume multiple members in gzip file except for record oriented I/O */

if (memcmp(magic, GZIP_MAGIC, 2) == 0
    || memcmp(magic, OLD_GZIP_MAGIC, 2) == 0) {

    method = (int)get_byte();
    if (method != DEFLATED) {
        fprintf(stderr,
            "%s: %s: unknown method %d -- get newer version of gzip\n",
            progname, ifname, method);
        exit_code = ERROR;
        return -1;
    }
    work = unzip;
    flags = (uch)get_byte();

    if ((flags & ENCRYPTED) != 0) {
        fprintf(stderr,
            "%s: %s is encrypted -- get newer version of gzip\n",
            progname, ifname);
        exit_code = ERROR;
        return -1;
    }
    if ((flags & CONTINUATION) != 0) {
        fprintf(stderr,
            "%s: %s is a multi-part gzip file -- get newer version of gzip\n",
            progname, ifname);
        exit_code = ERROR;
    }
}
```

Lock view with: 

History: <BACK|FORWARD>

To view the contents of a file, a player just walks into a file. When they are close enough, the file is displayed on the screen, as shown above, in a scrollable box. The file may be scrolled using the arrow and page up and down keys. The current position in the file is indicated by a scrollbar on the left and as line and column numbers at the top of the screen. The scrollbar cannot be currently dragged with the mouse, like a normal windows scrollbar, but this feature may be added in a future version of the tool.

The displayed text is shown using syntax highlighting, for easier comprehension. Function invocations and uses of variables are displayed as hyperlinks – by being underlined in blue. Clicking on a hyperlink will take the user to where the symbol in question is defined in the source code. If the definition is within the same file, the file is scrolled so that the definition is at the top of the screen. If the definition is in another file, the file view is closed, so that the user sees the 3D world. The user's view is then slowly panned around so that the file in which the symbol is defined is centred on the screen. The player is then slid towards the file, and upon reaching it the file is displayed, with the definition of the symbol at the top of the screen. In the present implementation, if the files are in different rooms separated by a wall, the player will be slid through the wall, which may be disorientating to the player. In future, part of the AI route-finding algorithm in Quake 3 could be reused to have the player follow a path to the destination file without crossing any walls.

Each time a player clicks on a hyperlink, the currently viewed file, and their position therein, is added to their history. The history operates exactly like a web browser's history, via a back and forward button.

5.4 Multi-User Features

Most of the multi-user features present in Quake 3 remain unaltered. These include the ability of players to see other players in the 3D world, to chat with them, and to view the names of all players in the current game by pressing the F1 key (the scoreboard key). In Quake 3, each player can choose a 3D model that will be their avatar in the 3D world. Each player also can set their name – this is the name that will be displayed in the scoreboard and in the chat dialog.

To chat in Quake 3, the user presses the talk key (usually t). They can type in a message, and when they press enter, the message will be printed in the top left hand corner of all the player's screens (accompanied by a short "beep" sound). This does not work if the player is currently using an in-game menu – in this case, any incoming messages are ignored. Since in the current implementation the source file contents display is implemented in the UI QVM, this means that players can't easily chat while viewing files. Chatting in this case is still possible, but in a roundabout way: pressing the tilde key brings down the console, where users can type messages to each other (messages beginning with \backslash are treated as commands, so message to be sent to other players must not begin with a \backslash). The disadvantage of this is that unless they have the console

open, they will not be notified of any incoming messages. This could be improved in future versions, by providing a chat box in the UI.

The standard Quake 3 interaction provides a simple way of seeing what file another player is looking at – one can walk around the map, looking for that player’s model, and see what file they are standing at. In future versions, an “overview” map could be added that displays, in 2D, the location of all files and players – this would simplify finding players on a large map. Unfortunately, the standard Quake 3 multi-user system does not provide a simple way of seeing which part of the file another user is looking at, and also there is no simple way of pointing out certain lines of code to another user.

Therefore, two new multi-user features were added to Quake 3 for this tool: a way of “locking” one’s view with another player’s, so that your screen shows exactly what the other player is seeing; and a way of “drawing” on the file display, so that one can point out parts of the source code to other users.

The lock view feature works as follows: when viewing a file (that is, when standing very near it in the 3D world), portraits of all other players also viewing the same file are displayed at the bottom of the screen. Clicking on one of these portraits locks your view with that player’s. This is indicated by the background of that player’s portrait flashing. When your view is locked with another player’s, you cannot scroll, but your view shows exactly what the other player sees – if they scroll, your view also scrolls to the same position.

One can point out bits of source code to other users currently viewing a file. This is done by dragging the mouse cursor with the right mouse button held down. This leaves behind a long trail of squares, which fade out over time. This can be used for example for circling or underlining certain items. In the present implementation, if one player viewing a file draws on it, the same trail of squares appears in the view of all other players also viewing the source file, irrespective of whether their views are locked or not. Whether the trail drawing should be restricted to only the players with locked views is something to be investigated in the future. Each player has a unique colour, which is used as the background of their portrait, as well as for the colour of the trail of squares. This is so users can easily identify who is drawing.

5.5 Detailed Design and High-Level Overview of Implementation

There are two distinct problems that need to be solved in the implementation: first, the parsing and analysing of the source code to create the hyperlinks and syntax highlighting information; and second, the actual display of these files in the 3D multi-user environment. This section provides a summary overview of the way these problems were overcome, while the next section, Implementation, contains a more detailed description.

5.5.1 Source Code Analysis and Cross-Referencing

The source code of the project to be visualised needs to be processed before it can be displayed. In this step, hyperlinks in the source code need to be created whether a symbol is used, with the target of the link being the definition of the symbol. Additionally at this stage mark-up for syntax highlighting is added. In principle, all of this processing could be done within the code running on the Quake 3 Virtual Machines, but due to the fact that these VMs support no dynamic memory allocation, are slower than native code and have limited file access ability, it was decided to do this processing before the tool is run.

In the present implementation, the pre-processing step consists of first running the doxygen[30] source code documentation tool, which generates a set of HTML pages of documentation and cross-referenced source code. Then, a purpose-written tool, temporarily named “dox2html”, is run with the output of doxygen as its input. This tool simplifies the HTML pages, removing style sheets, line numbers, and all of the unnecessary documentation, leaving behind only source code files with syntax highlighting and hyperlink information. The details of the output format are discussed in the implementation section.

The dox2html tool, after processing all of the files from Doxygen, also generates a file called `filenames`. This file contains the names of all files in the project, one per line. It is used as an integer-to-filename mapping by the code running in Quake 3.

5.5.2 Quake 3 Modification

The main modification to Quake 3 is the addition of a new entity, a `sourcefile` entity. Each source file entity has a unique identification number, `itemid`. The corresponding line in the `filenames` file (mentioned above) contains the name of the real source file represented by this entity.

Q3Radiant is the tool used to make and edit Quake 3 maps. In the current implementation of this project, this tool is used to modify an existing Quake 3 map, removing all of the normal game play items, and placing `sourcefile` entities around the map. Within Q3Radiant, unique numbers, starting at 1, are assigned to the `itemid` field of all the sourcefiles. In future, it may be a good idea to automate the generation of the map, by for example analysing the couplings between source code files and placing closely coupled source files next to each other.

Most of the code added to Quake 3 deals with handling various player interactions with the new `sourcefile` entity type. Modifications to code running on all three Quake virtual machines need to be made.

The `game` QVM (server) needs to be modified so that the `sourcefile` entities may be loaded at game start from the map. Additionally, the server keeps track of which players are using which files, for the lock view feature. The server also handles passing messages between players who have their views locked, to keep them synchronised. The server is also responsible for keeping track of any source files being carried – to make sure that they disappear when picked up and reappear when dropped. When a player walks up to an item, the server sends a message to the `cgame` QVM running on that player's computer, which forwards it to the UI QVM there so that the file is displayed. The server also enables players to “teleslide” between files when they click on hyperlinks. Upon receiving a command from a UI QVM indicating that player needs to be slid to a new file, the server animates that player's view to point at the new file, and starts sliding the player towards that file. If there is an intervening wall between the player and the destination, the server allows the player to go through the wall (this is called “noclip” in Quake 3). Another related modification is to change the `contents` type of player entities from `CONTENTS_BODY` to 0. This has the effect that players can run (and shoot) through each other, reducing crowding around files, and avoiding having a telesliding player end up stuck inside another player.

The `cgame` QVM (client) needs to be modified to display the file name labels above the source files in the 3D world. The labels are not drawn as part of the 3D world, but rather superimposed on top of the rendered view. This is done as it is easier to implement, and also it has the advantage that the labels are a constant size and always facing the player. There are some other modifications to the client, mostly to do with passing messages from the server to the UI, as the server cannot directly send a command to the UI of a specific client in Quake 3.

The UI QVM contains the biggest addition. Upon receiving a message from the server (forwarded to it by the client), the UI brings up a display of a source file. The UI handles the scrolling of the file locally, however it sends messages to the server to tell it what position the user is looking at, so that the server may forward this to other clients who have their view locked with this player. The UI also contains functions to keep track of the history, and to allow users to “draw” on top of the source file and forward the “drawings” to the other users of the same file, via the server. The text for the source files, along with the `filenames` file is stored locally on each player’s computer, in the `SRCVIS` directory within the main Quake directory.

One other minor modification needs to be made to both the `game` and `cgame` VMs – the removal of gore. Quake 3 by default draws large amounts of blood when players are shot or killed, and the death animations may be disturbing to some users. Most of the gore can be disabled by setting the `com_blood` cvar to 0. There is also no need for players to be running around with guns when viewing source files, so these were disabled from being drawn in the client code. Additionally, the map on which the current demo is based, `q3dm1`, had an animated pool of blood, which was removed.

Players are given an infinite amount of ammunition, which they need since files are picked up by shooting at them.

6 Tool Implementation

6.1 The *dox2html* Tool

The purpose of this tool is to take output from the doxygen[30] source code documentation tool, a set of HTML pages, and to remove extraneous information and complicated HTML elements, outputting files in a format that will be simple and fast to parse within the Quake 3 VM environment.

This tool is implemented in 420 lines of C++, in the file `dox2html.cpp`. The tool as written will only run on windows, but should be easy to port. The code is reasonably well documented, so the reader should refer there for any detailed implementation details. For the purposes of the present discussion, only the output format of this tool will be discussed in detail. For an example of how to use this tool see Section 7.3.

One file of output is created for each initial source code file (Doxygen creates many additional files, which are analysed by *dox2html*). Despite the name, the output of the tool is not actually valid HTML, but it is a very simplified form of HTML. There is no header or footer. For the most part, the output contains a verbatim copy of the original source file. Line breaks are kept in the same place as in the original. The normal HTML escaping of `&`, `<` and `>` as `&`, `<` and `>` exists in the output. Two HTML tags are placed within the output files by the *dox2html*.

The first tag is for links. It takes the form of

```
<a href="targetfilename#targetlinenumber">text</a>
```

much like a normal HTML link. However, the anchor points (the number following the #), are not explicitly declared inside the target file, as one would expect from standard HTML. Rather, the Quake 3 mod parses this number, then goes to target file, and then scrolls to the line number.

The second tag is for syntax highlighting. Doxygen uses a CSS-based system for syntax highlighting, so the tags it puts in the source files are just left in place by *dox2html*. These tags look like `text`. *Classtype* is one of: `keyword`, `keywordtype`, `keywordflow`, `comment`, `preprocessor`, `stringliteral` or `charliteral`. Each of these has a corresponding colour and style defined in the CSS file by Doxygen. Similar colours to these are used by the Quake 3 mod to display the syntax highlighting.

Currently, “dox2html” is incomplete, and some links in the resulting files are “dead” (have an invalid or null target), and some links in the original documentation are omitted (specifically, the ones that have to do with structs in C, since Doxygen uses a special format for documenting these, which dox2html currently just skips over.) This something that could be improved in the future.

6.2 The Quake 3 Modification

This section attempts to give a detailed overview of all the modifications made to the Quake 3. This section is written with two purposes in mind: to help future coders extend the present implementation of the tool, and to allow other people to judge the difficulty of using Quake 3 for information visualisation. This, after all, is the main aim of this project. This section was written after the modifications were made, so there may be some omissions or minor factual errors below. All modifications or additions were tagged with a `\\bkot` comment, so searching the source code for `bkot` should bring up all the modifications and additions.

The sections below document the modifications, one file at a time, to the Quake 3 game logic code running on the three Quake 3 Virtual Machines.

6.2.1 Server-side Modifications (game.qvm)

6.2.1.1 Modifications to `bg_public.h`

This file contains declarations of several data structures types that are used by both the client and server (“bg” refers to Both Games).

- ◆ A new `#define` was added, `#define PMF_FRICTIONLESS 32768`. This is used in the `pm_flags` field of the `pmove_t` struct. PMF refers to Physical Move Flag. There are some existing PMF_s, such as walking, flying and swimming. `PMF_FRICTIONLESS` was added as this is the type of movement the player undergoes when they teleslide to a file. None of the existing PMF_ types provided the necessary behaviour for telesliding (moving at a constant velocity, through walls).
- ◆ A new element was added to the `holdable_t` enum, `HI_SOURCEFILE`. “HI” refers to Holdable Item. This is used by the code to keep track of what type of item the player is carrying.

6.2.1.2 Modifications to `g_local.h`

This file contains declarations only used by the server code (“g” is for “game.qvm”)

- ◆ One function declaration was added: `Drop_Sourcefile()`. This function is used when a player wants to drop a sourcefile they are carrying.
- ◆ Two fields were added to the `gentity_s` struct. One instance of this struct is used by the server to keep track of the state of each entity in the game. The added fields are only guaranteed to have valid values if the entity is a `sourcefile`. The added fields are:
 - `int itemid` This is a number which is unique for each sourcefile. It corresponds to the `itemid` field set in Q3Radiant, the map editing tool for Quake 3. The `itemid` line in the `filenames` file contains the filename of the file represented by this sourcefile entity.
 - `int users` This is a set of 32 bits, one for each of the first 32 clients to join the game. If bit number n is set (LSB=0), this indicates that player n is currently “using” this entity, that is that they have walked up to it and are viewing the contents of the file.
- ◆ Thirteen fields were added to the `gclient_s` struct. One instance of this struct is used by the server to keep track of the state of each client (player) in the game.
 - `int using_entity` This stores the index in the global `g_entities` array (in which all entity data structures are kept) of the entity that this client is currently using (viewing). -1 indicates the client is not viewing any. This information is redundant with the `users` field of the entity struct, but is kept here to speed up the passing of scroll and draw events between users of the same file.
 - `int touch_sourcefile_frame` This stores the game time when this client last touched a source file entity. This is needed to avoid repeatedly drawing the contents of a sourcefile when a player walks into a sourcefile and stops. A message is only sent to the UI if the user touches a file, and the client `touch_sourcefile_frame` is sufficiently far in the past (more than 0.5 seconds).
 - `int target_item` When the client is telesliding, this stores the `itemid` of the destination `sourcefile`. This is needed to avoid displaying the contents of any sourcefiles that the client may touch while sliding to the destination file.
 - `int target_linenum` When telesliding, this stores the line number to be displayed when the client reaches the destination.
 - `float target_angles[3], float target_angles_start[3], int target_angle_time` These three fields are used for smoothly panning the

- client's view toward the destination `sourcefile` when starting a teleslide. Ideally, this could all be done in the client (`cgame.qvm`), but doing it on the server appeared easier.
- `float target_dir[3]` This stores a vector pointing in the direction of the destination `sourcefile` when telesliding.
 - `int followers` This is a set of 32 bits, one for each of the first 32 clients to join the game. If bit number n is set, this indicates that that player is currently “following” this client. This is used for forwarding scroll messages to players with locked views.
 - `int following` Indicates the `clientid` (index into the `g_entities` array) of the player the client is following, -1 if none.
 - `int scroll_last_line, int scroll_last_col` These record the last scroll event received from the client. This should correspond to the place where the client is currently looking inside the `sourcefile`. This is needed so that when another client locks his view with this player's, the server can tell the locking client where to scroll to, without asking the other player for an update of its scroll position.
 - `gentity_t *item_ent` Pointer to the entity which the player is carrying, that is the one that was shot by the player, NULL otherwise. This is used as follows: when the player shoots a `sourcefile`, it is made invisible, and a pointer is assigned to this field. When the user drops the file, the entity is moved to the current player location, and made visible. This is the normal mechanism for holdable items implemented by Quake 3.

6.2.1.3 Modifications to `bg_misc.c`

This file contains miscellaneous function and variable definitions use by both the server and client.

- ◆ A new item type, `sourcefile`, was added to the `bg_itemlist` array. This is where the connection between the type of entity and its display model is made. For now, `sourcefile` uses the same model as the holdable teleporter. Additionally, `sourcefile` is defined to have a `giTag` of `HI_SOURCEFILE` and a `giType` of `IT_HOLDABLE`. This provides a way of distinguishing `sourcefiles` from other entities when processing the `g_entities` array.
- ◆ The `BG_CanItemBeGrabbed()` function was modified. This function tests whether a player should be able to pick up an item – in Quake 3, when a player walks over an item the item is picked up. The modification was that if the item is a `sourcefile`, the function returns `qfalse`, since `sourcefiles` are picked up by shooting them, not walking over them.

6.2.1.4 Modifications to `bg_pmove.c`

This file contains functions used by both the client and server which define the physics of player movement.

- ◆ The function `PM_NoClipMove()` was modified so that if the player's move flag `PMF_FRICTIONLESS` is set, no friction would be applied to the movement. This is to ensure that when telesliding, the player will move at a constant velocity.

6.2.1.5 Modifications to `bg_slidemove.c`

This file contains functions used by both the client and server which define the physics of player movement. Specifically, it contains functions that make players bump off walls they hit when sliding around.

- ◆ The function `PM_SlideMove()` was modified so that if the player's move flag `PMF_FRICTIONLESS` is set, the player didn't bump off any walls. This was necessary since a teleslide often takes the player through a wall.

6.2.1.6 Modifications to `g_active.c`

This file contains many of the message passing functions that handle events from clients and generate events in the server, such as deciding when a player has touched an item.

- ◆ `ClientEvents()` was modified, to respond to the `USE_ITEM` event from the client, which is generated when the player presses the use key (usually `e`). If the player is carrying a `sourcefile`, it will be dropped by involving the `Drop_Sourcefile()` function.
- ◆ `ClientThink_real()`, which is called once for each client animation frame, was modified to create the smooth panning of the client view towards the destination when starting a teleslide. The modification makes use of the `target_*` fields of the client, mentioned above.
- ◆ `ClientThink_real()` was additionally modified so that if the client is moving with `PMF_FRICTIONLESS` set the client can touch items, even if the client's `noclip` flag is set. This is so touch item events are generated for `sourcefiles` even when the player is telesliding (this is not normal Quake 3 behaviour) – so that the server knows when the player reached the target.

6.2.1.7 Modifications to `g_client.c`

This file contains functions that deal with clients joining and leaving the game.

- ◆ `ClientSpawn()`, the client set-up function, was modified so that all the clients' `r.contents` fields are set to 0 – this means that players can walk and shoot through each other. This is to avoid players becoming stuck in a crowd around files, and to reduce distractions from bored users walking around and shooting other users. The function was also modified to give the clients infinite ammo for the machinegun weapon, and to set all the clients' god flag, making them invulnerable.
- ◆ `ClientDisconnect()` was modified so that if a client leaves the game while they are carrying a `sourcefile`, it is dropped, rather than vanishing off the map.

6.2.1.8 Modifications to `g_cmds.c`

This file contains handlers for all commands sent by the clients. Commands are messages consisting of a variable-length, null-terminated string of characters. The strings consist of space-delimited tokens. The first token is the command type, and the following tokens are command arguments. Four new command types and handlers were added.

- ◆ `ClientCommand()` was modified to invoke the appropriate new handlers when one of the four new commands, `sv_gotofile`, `sv_following`, `sv_scrolled` or `sv_trail` was received by the server from a client.
- ◆ A new command handler `Cmd_following()` was added. This command is sent by a client when that client wants to lock its view with some other client's. This command takes the form of `sv_following who`, where `who` is the `clientid` of the other client. If `who` is -1, the server records that the client is no longer following anyone. The handler updates the `followers` and `following` fields of the clients involved. It then sends a `cg_scrolled` command to the initiating client, telling it where to scroll to synchronise its view.
- ◆ A new command handler `Cmd_GotoFile()` was added. This command is sent by the client when it wants to teleslide to another file, for example due to the user clicking on a hyperlink. The command takes two arguments, the destination `itemid` and the destination line number. The function locates the destination file, and sets up the fields in the client structure so the various telesliding animations can happen. Additionally, it informs all followers of this client that it is telesliding. The details of this function are best understood by looking at the source code of the handler, due to many special case scenarios. Telesliding players are made invisible, since if there are two or more players telesliding together to the same destination,

they will occupy the same physical location, which results in “jittery” drawing on the clients’ screens. This jitter may be an effect of the client-side prediction failing to deal with players with `r.contents` set to 0 correctly, however making the players invisible is an acceptable solution for now. Players can slide at two speeds: if the destination is far away, the player moves much faster towards it than if it were nearby. This is to avoid the player getting lost when going a short distance, by giving them time to orient, but on the other hand it avoid lengthy waits when going far. In future, perhaps the speed of the player could be controlled by the user via accelerate and decelerate keys.

- ◆ A new command handler `Cmd_scrolled()` was added. This is responsible for forwarding scroll messages from each client to all the clients that have their views locked with that client. This is done by referring to the `followers` field of the initiating client’s data structure. The handler takes two arguments, the current scroll line position and scroll column position. The handler sends messages to all followers in the format `cg_scrolled clientid line column`, where `clientid` is the id of the initiating client.
- ◆ A new command handler `Cmd_trail()` was added. This is used for forwarding drawing messages between all players viewing a `sourcefile`. When the user at one of the clients drags the mouse with the right mouse button held down, the client sends a `sv_trail x y` command to the server every few hundred milliseconds, where `x` and `y` are the current mouse position. The `Cmd_trail()` handler forwards these messages to all other clients who are currently using the same file as the initiator. The forwarded message is of the format `cg_trail clientid x y`, where the `clientid` is that of the initiating client – so that the receiving client knows what colour to draw the trail with.

6.2.1.9 Modifications to `g_items.c`

This file contains most of the functions that deal with interacting with items in the game, such as picking them up, shooting and using them, as well as the code that initialises (“spawns”) the items in the game.

- ◆ The `Touch_Item()` function was modified to specify what happens when a user walks up to a `sourcefile`. The modification first checks if any work needs to be done. Quake 3 triggers the `Touch_Item()` function continuously when a player is within the bounding box of an item. For the purposes of this project, only one event is needed when the user first enters the bounding box. This is approximated by recording the time that the last `Touch_Item()` event occurred in the `touch_sourcefile_frame` field of the client data structure. If upon entering `Touch_Item()` this time is more than 500ms in the past,

this is taken as meaning the user just recently entered the sourcefile bounding box. Only in this case is the rest of the modified code run. The `touch_sourcefile_frame` is updated with the current game time (`level.time` global variable) every time `Touch_Item()` is invoked for a `sourcefile`. This whole solution is not ideal, since sometimes the server may pause for more than 500ms, for example if a client running on the same machine as the server needs to load a large file, which causes the modified code to be run again. In future, a better solution for this needs to be found, but the present solution works acceptably on reasonable fast computers.

The bulk of the modification deals with stopping the player if they are telesliding, updating the `users` and `using_entity` fields of the entity and client structs, telling the client that it needs to bring up a display of the `sourcefile`, and letting other users of the same file know that a new user has joined them. The details of this are best understood by referring to the source code.

- ◆ `Touch_Item()` was further modified to prevent the standard Quake 3 item pick up function being invoked when a player touches a `sourcefile`.
- ◆ A new function, `Drop_Sourcefile()` was added. It is a customised version of `Drop_Item()`. This function handles moving the dropped `sourcefile` to the correct position, and making it visible.
- ◆ `FinishSpawningItem()` was modified, to set the `contents` type of `sourcefiles` such that they can be shot at. In normal Quake 3, items cannot be shot – bullets go through them.
- ◆ A new function, `G_SourcefileHit()` was added. This is used as callback function in source file's entity data structure, which is called by Quake 3 when the `sourcefile` is hit by a bullet. The callback checks if the attacker is already carrying a file and, if not, hides the item from view on the map and adds a pointer to it to the client's `item_ent` field.
- ◆ `G_SpawnItem()` was modified, to initialise the `sourcefile`'s entity data structure so that the `G_SourcefileHit()` callback is called when the entity is shot. Additionally, the `sourcefile`'s `itemid` is copied to that entities' `s.generic1` field. The `s` member, of type `entityState_t`, contains all the state information of the entity which is sent via the network to the client. Since the client needs to know each `sourcefiles`' `itemid` (so that it can display the correct filename and file contents), the `itemid` must be put somewhere into the `s` member. Unfortunately, `entityState_t` cannot be modified, as it is directly accessed by the Quake 3 executable (game engine), whose source code is not available and so

cannot be altered. However, within `entityState_t` there is a rarely-used field, `int generic1`, which can be made to do double-duty as the `itemid`. This is a good example of a limitation of using Quake 3 – the network protocol cannot be altered. This is not a fatal limitation, since firstly the `generic1` field exists, and secondly a workaround could be coded where extra state information is passed via server and client commands.

6.2.1.10 Modifications to `g_spawn.c`

This file contains functions and data to do with starting the game world, and placing all entities in their start states.

- ◆ An item was added to the `fields[]` array. This array keeps track of what fields in the entity structure (`entity_t`) are to be loaded from the map file. The item added was `itemid`, with a type of integer (which is need for correct parsing the map file). This modification was needed so that the `itemids` specified in Q3Radiant were correctly loaded into the server's entity state data structure.

6.2.1.11 Modifications to `g_utils.c`

This files contains miscellaneous server functions.

- ◆ The function `G_KillBox()` was modified so it immediately returns – so that is does nothing. This function is normally used by Quake 3 when a player enters the game world, to make sure no players exist at the start point. This is to avoid players becoming stuck in each other. Since in the implemented tool, the `contents` type of players was changed so that the players can move through each other, this is no longer needed.

6.2.2 Client-side Modifications (cgame.qvm)

6.2.2.1 Modifications to cg_local.h

- ◆ A field was added to the `centity_t` structure, `float size`. This is used for storing the size at which sourcefiles should be drawn. This value is initialised according to the size of the file represented by the `sourcefile` entity.

6.2.2.2 Modifications to cg_draw.c

- ◆ The function `CG_DrawScores()` was disabled (returns instantly). In a normal Quake 3 game, player accumulate kills or “frags” – the player with most frags wins. This function displays the current player’s and leader’s score in the bottom-right-hand corner of the client display. This is not needed, so was disabled
- ◆ The function `CG_DrawAmmoWarning()` was likewise disabled. This function normally displays a warning when the player is low on ammunition. This warning is incorrectly displayed even when the player has infinite ammo, so it was disabled.
- ◆ `CG_Draw2D()` was modified, to call `CG_Sourcefiles()` to draw the names of sourcefiles.

6.2.2.3 Modifications to cg_ents.c

This file contains code that deals with entity animations and spawning.

- ◆ The function `CG_Item()` was modified, to alter the way `sourcefile` entities are drawn. Source files are scaled in all directions by the value of their entity data structure’s `size` field. If the file is a header file (ends in `h`), a sphere is drawn around the `sourcefile` entity.

6.2.2.4 Modifications to cg_events.c

- ◆ Minor change to `CG_UseItem()`, so that no message is displayed when a `sourcefile` is dropped. Also, if the player attempts to drop a `sourcefile` when none is being carried, a message is displayed saying “Not carrying any files!”

6.2.2.5 Modifications to `cg_main.c`

- ◆ Two function calls were added to the main client start up routine: `CG_InitMemory()` and `SRCV_Init()`.

6.2.2.6 Modifications to `cg_servercmds.c`

This file contains code that responds to commands sent by the server to the client.

- ◆ In `CG_MapRestart()`, the drawing of a “FIGHT!” message was disabled. This is normally displayed when the game starts.
- ◆ `CG_ServerCommand()` was modified, to recognise three new commands: `cg_show_file`, `cg_scrolled` and `cg_trail`. These commands are sent almost verbatim to the UI QVM, where they are processed. The code in this function replaces the `cg_` prefix with `ui_`, so that `cg_trail` becomes `ui_trail`. This is to avoid an infinite loop, since when the client sends any command, it is first processed by `CG_ServerCommand()`. If the name were not changed, the client would keep sending the same message to itself forever. (At least, that is what appeared to be happening during debugging.)

6.2.2.7 Modifications to `cg_snapshot.c`

This file contains functions which are called at each client calculation frame (render frames may occur more often).

- ◆ `CG_ResetEntity()` was modified, to initialise the size field of sourcefiles correctly, based on the corresponding file size. This is probably not the best place to do this, since it will be done more often than necessary when called here. In future, a better place needs to be found.

6.2.2.8 Addition of `sourcevis.c`

This new file was added to handle two tasks: the mapping of `itemids` to real filenames, and the drawing of the file names above the corresponding `sourcefiles`.

- ◆ The mapping is carried out as described in the design section, where line number n of a file called `filenames` contains the name of the file corresponding to the `sourcefile` with `itemid` of n . The smallest allowed value of `itemid` is 1.

- ◆ The drawing of the filenames is based on code taken from a web forum[31], which in turn took it from “HLHack 1.3 by deltashark.” Essentially, the code projects the 3D location of the `sourcefile` on to the 2D screen, moves up a bit, and draws the file name there. If the filename ends in `h`, the filename is drawn in red, else in blue. The main non-static function provided for file name drawing is `CG_Sourcefiles()`, which loops through all `sourcefiles` and prints their names. A side-effect of the client-server design of Quake 3 is that only the file names of source files which are potentially visible are drawn. This is because the server only tells the client about entities which are potentially visible to that client, to avoid cheating.

6.2.2.9 Modifications to `cg_weapons.c`

- ◆ `CG_MachineGunEjectBrass()` was disabled, so that no brass (empty shell cartridges from the machinegun) is drawn.
- ◆ `CG_RegisterItemVisuals()` was modified to correctly set up the sphere around header `sourcefiles`.
- ◆ `CG_AddPlayerWeapon()` was disabled, so that players are drawn without guns (see design section).

6.2.3 Modifications to the UI (ui.qvm)

6.2.3.1 Modifications to ui_atoms.c

- ◆ `UI_ConsoleCommand()` was modified, to add three new commands: `ui_show_file`, `ui_scrolled` and `ui_trail`. When these commands are received the functions `UI_SourceFile_Draw()`, `UI_SourceFile_ScrollCmd()` and `UI_SourceFile_TraceCmd()` are invoked, respectively. Arguments are converted from strings to integers where appropriate before invoking the functions.

6.2.3.2 Addition of ui_sourcefile.c

This file, 1200 lines long, contains all the functions that deal with displaying the source files' text, clicking on hyperlinks, scrolling, providing back and forward buttons, locking views, and communicating with the server (via the client). There are many complicated functions inside this file, so only a brief overview is given below, the source code remains the best place to learn about the detailed workings.

`UI_SourceFile_Draw()` is the function which is normally called first, via the `ui_show_file` command from the server. It loads the source file into memory, and sets up the UI QVM for drawing of the sourcefile (registers `sfMenuDraw()` as a draw callback for the current menu).

Due to the lack of dynamic memory support in the Quake 3 VMs, the file is loaded from disk in 50K byte chunks – only one chunk is kept in a (static) buffer at a time. When the user scrolls outside of the currently loaded 50K chunk, a new 50K chunk is loaded. This approach adds quite a lot of complex code. Theoretically, one could create a huge static buffer (say 100MB), and allocate memory out of that as needed. The problem with that is there seems to be no documentation regarding how much memory is accessible to the QVM – but there is a comments somewhere in the code that it must fit in the stack of the host CPU, so this is probably platform dependant. This might be something worth looking into in future versions. When the Quake 3 engine source code is released, some form of dynamic memory allocation could perhaps be add to the VMs.

One limitation due to the lack of dynamic memory allocation wasn't worked around: the `filenames` file is limited to 16K. This limit is plenty for all practical applications, but could be removed in a future version.

The normal Quake 3 UI functions do not provide a scrollable textbox. Therefore, all the scrolling and display functions had to be coded by hand. These new functions only use standard Quake 3 functions for drawing individual characters on the screen – all other functionality was coded from scratch. This shows that the Quake 3 UI is not very advanced, and may need a lot of work to tailor it to a specific application. However, some Quake 3 UI replacements and improvements are available on the internet – these may be investigated in the future.

The user can scroll the text by using the arrow keys, pageup and pagedown keys, and the mouse scrollwheel. The current position in the file is displayed as a scrollbar on the left side of the screen. Currently, the scrollbar is not fully implemented, so the user cannot click on it to scroll. This should be finished in a future version.

The source code file is parsed as it is loaded. When syntax highlighting tags are encountered (see Section 5.5.1), the current draw colour is changed to that corresponding to the syntax element. If a hyperlink tag is encountered, the following text is underlined in blue.

When the user clicks on the text, the code identifies what line was clicked, and then parses that line to see if there is a link at that position. If so, it identifies the target file and line number, and adds it to the history. If the destination is in the same file as is currently displayed, the view is scrolled there, otherwise the UI instructs the server (via the `sv_gotofile` command) to teleslide the player to the destination file.

The UI keeps track of what other players are viewing the same file as the UI is displaying, via receiving commands from the server. At the bottom of the screen, this information is displayed as portraits of all the other players. Each portrait's background colour is unique. When a user clicks on one of the portraits, its background starts flashing. The scrolling functions are then disabled, and a message is sent to the server to tell it that this user wants to follow another user. The server will then send messages whenever the other user scrolls or clicks a link, which are processed by the UI so as to keep the two views identical.

If the user drags the mouse with the right mouse button down, a trail of squares is drawn on the screen. Messages containing the location of the cursor are sent to the server, which will forward it to the UIs of all the other players also viewing the same file. When the UI receives such a message, a new trail is drawn at the specified location, with the colour corresponding to the portrait background of the user drawing the trail. At the moment, these trails are drawn even if the players are looking at different parts of the file. This should be fixed in future versions.

One major bug still remains, which can cause the client to crash when viewing a text file which fills the entire screen with text. It seems that in this case, Quake 3 reaches some internal limit for the number of polygons it can draw. When this happens, invoking the `r_speeds 1` command via the console shows that around 8200 triangles are being drawn on the screen. In practice, most source files have plenty of white space, so this bug does not occur often. However, this is a major bug, and needs to be fixed in future versions. This is possibly a bug of the Quake 3 game engine, so access to the Quake 3 game engine source may be needed to fix this.

7 Tool usage

7.1 Installation

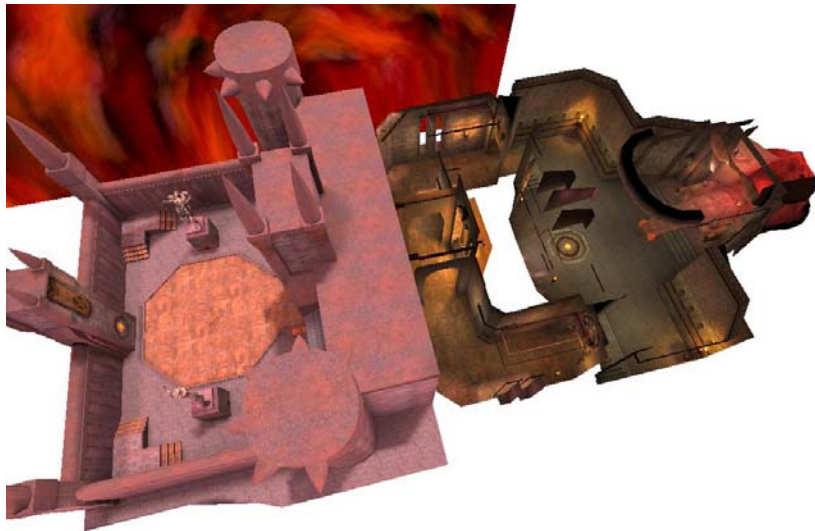
A working Quake 3 Arena install is needed to use this tool. To set up the tool and the demo, unzip the file `SRCVIS.zip` into the Quake 3 Arena directory. This should unzip several files and directories into a new directory called `SRCVIS`. This directory should be in the same place as `quake3.exe` and the `baseq3` directory. Running `srcvis.bat` from within the `SRCVIS` directory should start Quake 3, load the tool, and start the demo.

The player's name and model can be changed by pressing `Esc`, selecting `SETUP` and then `PLAYER`.

To set up the tool with several users, firstly run the `srcvis.bat` file as described above on all the computers. Choose one of the computers to be the server – this preferably should be the fastest computer available. There are then two ways to proceed: if the IP address of the LAN (or internet) network interface of the server is known, then on all the other computers, press `~` and type `\connect 192.168.0.13` without the quotes, replacing the address by the address of the server. If the IP address is not known, then on all the other computers go to the menu, press `Esc` and choose `LEAVE ARENA`. This will halt the servers running on these computers. Then, go to `MULTIPLAYER`. Hopefully, the one computer still running a quake server will be shown there. If not, try pressing `REFRESH`. Once the server is shown, click on it to highlight it and then click on `FIGHT` to connect. If one method doesn't work, try the other one. If both fail, try using a different computer as the server – Quake 3, and network games in general, often don't seem to run well on certain machines.

7.2 Example of Multi-user Code Walkthrough

This example will show how a multi-user walkthrough of some source code can be carried out using the implemented tool. The example code used for this walkthrough is that of the `gzip[32]` Unix command line compression and uncompression utility. This was chosen because its source code consists of twenty files, which seemed like a good amount for a simple example. The demo map is based on the `q3dm1` map which comes with Quake 3, shown below. It was chosen because it is simple, and its source (uncompiled form of the map) is provided by id Software. This map consists of two main areas: an outdoor courtyard and an indoor room, connected by corridors:



Two source files are shown below. On the left is `crypt.c`, only a few lines long, while `gzip.c`, which is over 100KB long is on the right. (The text labels may be difficult to read due to the reduced size of the image.)



Below is an overview of the courtyard and the files placed in it. On the left pedestal are `gzip.h` and `gzip.c`, the two main files in the project. Notice the header files are displayed in red. Nearer the camera are `util.c`, `getopt.c` and `getopt.h`. Together, these five file contain most of the argument parsing and file access functions.



On the right side are `crypt.h`, `crypt.h`, `lzw.h`, `lzw.c`, `revision.h` and `tailor.h`. All of these are very small files, which perform very little work in the program. Therefore, they were all put in the same part of the map.

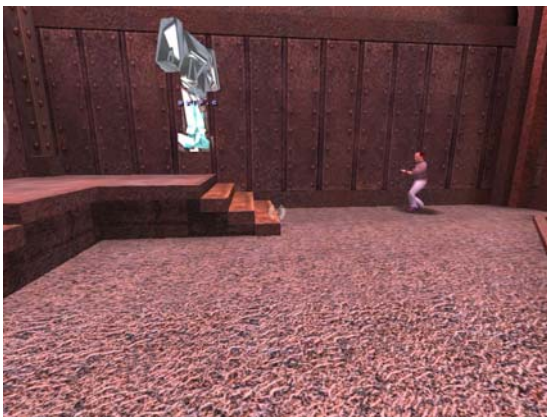


In the room there are nine files which contain implementations of the various compression and uncompression algorithms. On the right, near the camera is `unpack.c`, `unlz.c` and `unlh.c`. Further away on the right side are `zip.c` and `unzip.c`. On the left hand side are `deflate.c`, `inflate.c`, `trees.c` and `bits.c`.

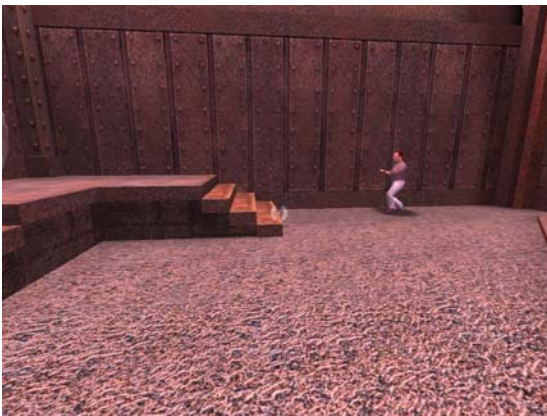
Let us introduce two players to demonstrate the multi-user features: Claus (left) and Tim (right):



Screenshots taken from Claus's perspective will be shown on the left, those seen by Tim will be on the right. Tim will first demonstrate how to move a file. Tim looks at the sourcefile he wants to move:

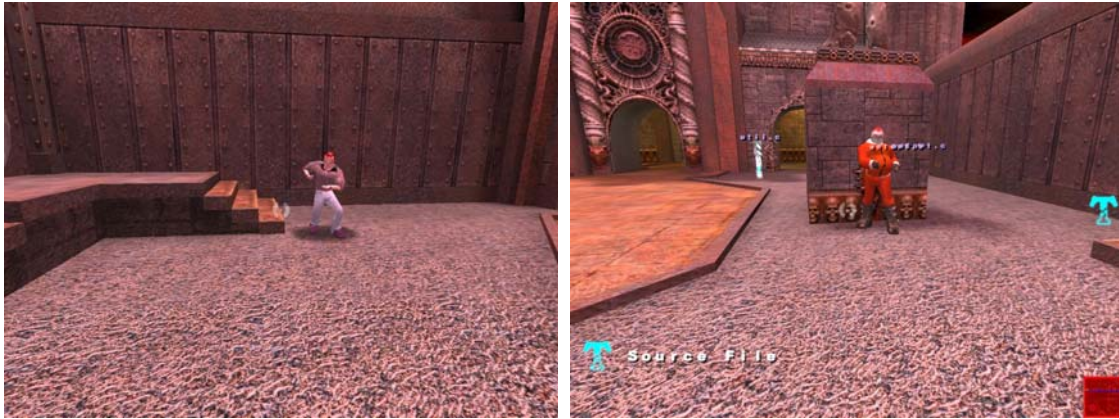


And then shoots:

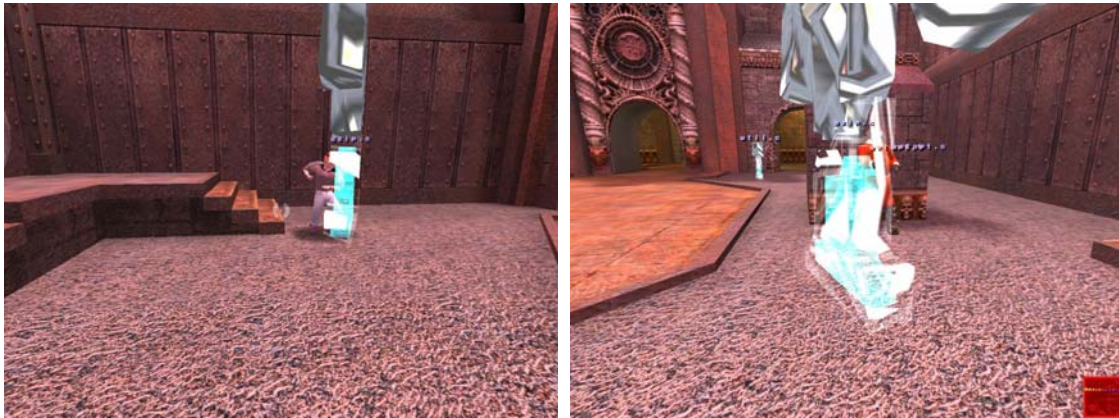


The file disappears, but a 'T' icon appears on Tim's display.

Tim moves to the place where he wants the file to go to (this could be anywhere on the map):

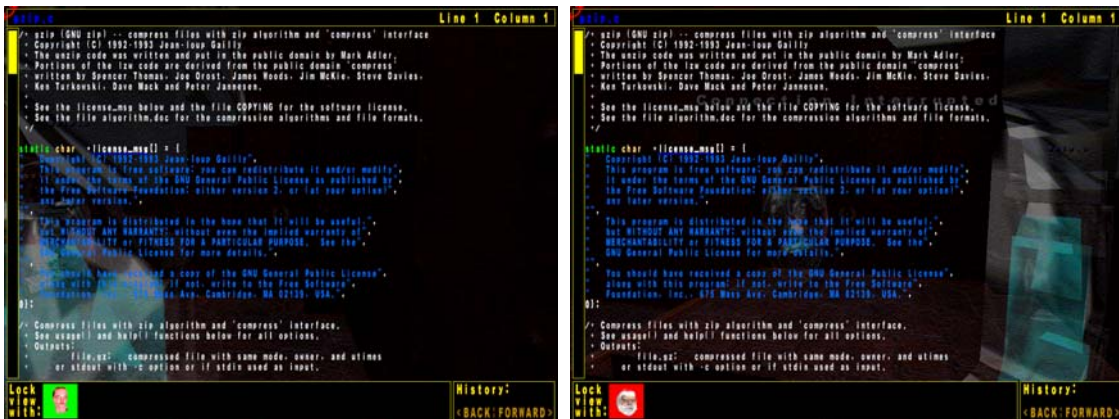


And then presses the use key (typically e) to drop the file:



The file appears at the new location.

Tim will now guide Claus through part of the source code of gzip. Both users go up to `gzip.c`:



They can now both see the beginning of the file. At the bottom of the screen, next to “Lock View With:” Claus can see Tim’s portrait and vice versa. Claus clicks on Tim’s portrait to lock his view with Tim’s.

The green background of Tim's portrait starts flashing (difficult to show on paper). Claus' view will now always follow the scroll position of Tim. Claus cannot scroll using the keyboard or mouse. Tim now scrolls to what he wants to show to Claus. Tim circles a part of the source file by dragging the mouse with the right mouse button held down. This leaves a trail of green squares on both player's screens. Since both screens are practically identical, only Claus' view is shown below:

```


gzip.c                                     Line 1168 Column 1
}
method = -1;                               /* unknown yet */
part_nb++;                                 /* number of parts in gzip file */
header_bytes = 0;
last_member = RECORD_IO;
/* assume multiple members in gzip file except for record oriented I/O */

if (memcmp(magic, GZIP_MAGIC, 2) == 0
    || memcmp(magic, OLD_GZIP_MAGIC, 2) == 0) {

    method = (int)get_byte();
    if (method != DEFLATED) {
        fprintf(stderr,
            "%s: %s: unknown method %d -- get newer version of gzip\n",
            progname, ifname, method);
        exit_code = ERROR;
        return -1;
    }
    work = unzip;
    flags = (uch)get_byte();

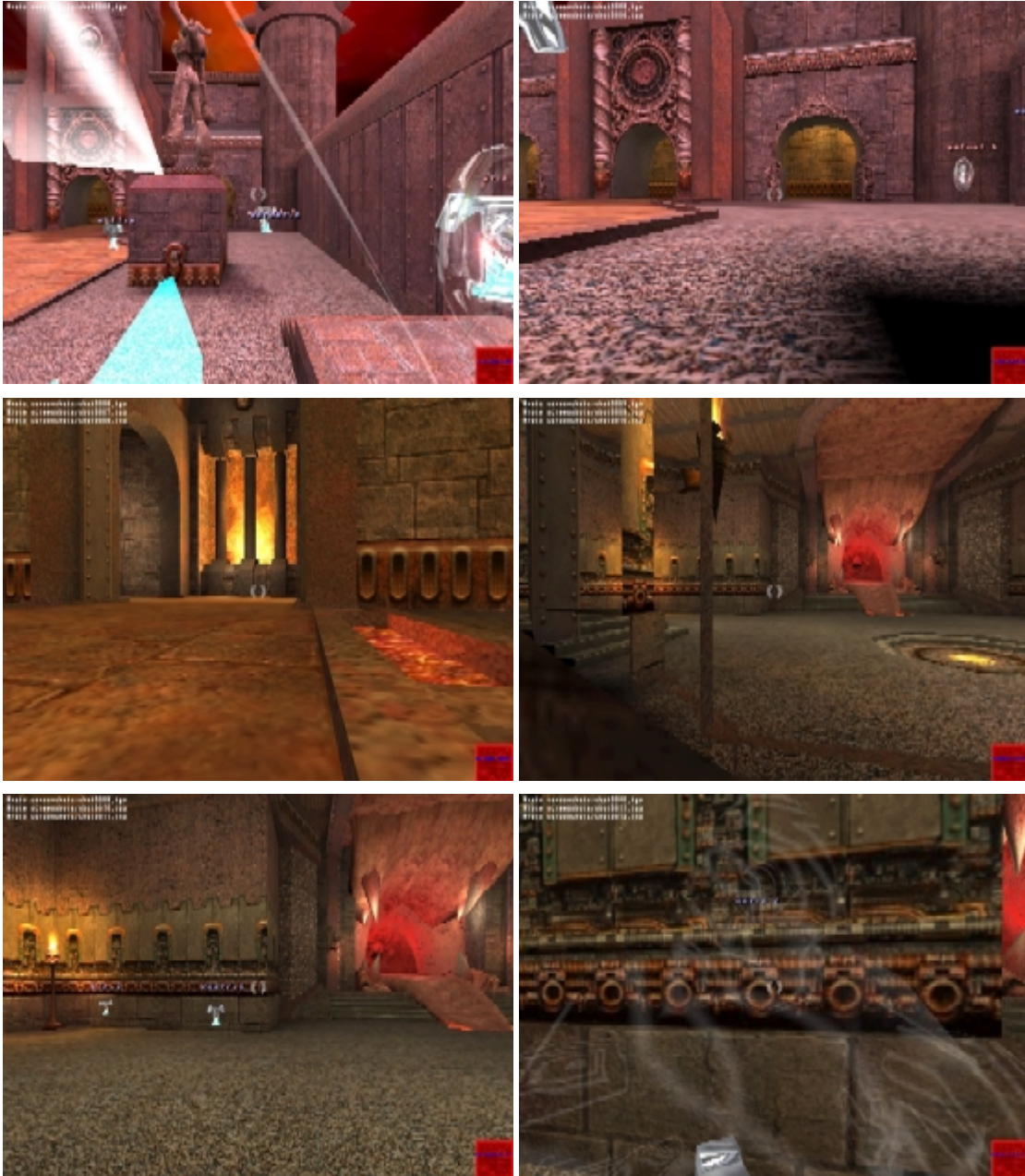
    if ((flags & ENCRYPTED) != 0) {
        fprintf(stderr,
            "%s: %s is encrypted -- get newer version of gzip\n",
            progname, ifname);
        exit_code = ERROR;
        return -1;
    }
    if ((flags & CONTINUATION) != 0) {
        fprintf(stderr,
            "%s: %s is a multi-part gzip file -- get newer version of gzip\n",
            progname, ifname);
        exit_code = ERROR;
    }
}

```

Lock view with: 

History: <BACK|FORWARD>

It can be seen that Tim is underlining the `unzip` hyperlink (the trail looks a lot better and cleaner when viewed for real.) Tim did this as he wants to show Claus he is about to click on it. When he does do so, both players will be slid to the file where `unzip` is defined. A few frames from this animation is shown below, in sequence left to right, top to bottom. Again, both players' views are identical, so only one is shown:



Upon reaching the destination, both players see the same view, of the definition of `unzip()`. This is shown below, this time from Tim's screen:

```

unzip.c Line 99 Column 1
int unzip(in, out)
int in, out; /* input and output file descriptors */
{
    ulg orig_crc = 0; /* original crc */
    ulg orig_len = 0; /* original uncompressed length */
    int n;
    uch buf[EXTHDR]; /* extended local header */

    ifd = in;
    ofd = out;

    updcrc(NULL, 0); /* initialize crc */

    if (pkzip && !ext_header) { /* crc and length at the end otherwise */
        orig_crc = LG(inbuf + LOCCRC);
        orig_len = LG(inbuf + LOCLEN);
    }

    /* Decompress */
    if (method == DEFLATED) {
        int res = inflate();

        if (res == 3) {
            error("out of memory");
        } else if (res != 0) {
            error("invalid compressed data--format violated");
        }
    } else if (pkzip && method == STORED) {
        register ulg n = LG(inbuf + LOCLEN);
    }
}

```

Lock
view
with:

History:
< BACK | FORWARD >

Tim could now click on the `inflate()` link to go and look at the implementation of the unzip algorithm. He then could click on `BACK` to come back to the view above. Or, he could scroll up or down to see other methods in the file. He could also press `Esc`, to close the file view, and then walk to some other file in the world. In that case, Claus' view would be unlocked, and Claus would be able to scroll normally again.

The players can currently chat in two ways: if a file is not being displayed, either player can press the talk key (usually `t`), and type their message. This will appear on the other players' screens, with an audible alert. Otherwise, if a file is being displayed, the players will need to drop down the console to chat by pressing `~`. They can type messages to each other, as long as they don't start with `\` (since these will be interpreted as commands). This chatting-via-console is not optimal, since other players are not notified when a messages arrives for them. Improving the chat system in the file view is a planned as a future improvement.

This demo only had two players. Quake 3 supports up to 64 players, but some limitations in the implemented tool (using 32 bit for the `users` field) mean only 32 players maximum can use the source files.

7.3 Setting up the Tool to Visualise a Different Source Project

The process of setting up the tool to visualise a source project is rather complicated, so it is outlined here, using the same example, `gzip`, as above.

Download and install `doxygen`[30]. Start the ‘doxywizard’ program. Choose the ‘Wizard’ button. Type in `gzip` for the Project name. Select the source directory containing the `gzip` source code, and create a new directory and set it as the destination directory. On the ‘Mode’ tab select ‘All entities’ and ‘Include cross-referenced source code in output’. Then select ‘Optimize for C output’. On the ‘Output’ tab select only HTML output, and make sure ‘plain HTML’ is selected. On the ‘Diagrams’ tabs select ‘No Diagrams.’ Click the OK button

In the main window, under Step 2, click on ‘Save’ and save the configuration file somewhere. Under Step 3, specify the working directory; the directory containing the source files seems to work well. Click on the start button. After a few minutes, `doxygen` should finish.

Now copy the `dox2html` executable into the directory which was the destination directory for `doxygen`. Run it from there (`dox2html` needs the `doxygen` output to be in the same directory as itself.) A console will pop up, and lots of messages will scroll past – these indicate links not processed due to some features not being implemented in `dox2html` yet. After a few minutes (`dox2html` is not particularly fast) the tool should finish. If all went well, a new directory `out` should have been created, containing all the source files and a `filenames` file. Move or copy these into the `SRCVIS` directory under the Quake 3 directory.

At this point, the map may need to be edited to alter the number or location of `sourcefile` entities in it – this can not be done using the implemented tool at this stage, so `Q3Radiant` will need to be used. This is quite a complex tool, so refer to its documentation to learn how to use it. The file to load is `testmap2.map` from the `SRCVIS\maps` directory. Add or remove sourcefiles like you would any other entity. To assign `itemids` to sourcefiles, select the `sourcefile`, press `n` to bring up the entity window, and type in `itemid` in the key field and a number in the value field. Press `enter` to store the new value. The map needs to run through the normal Quake 3 `bsp`, `vis` and `light` toolchain to generate `testmap2.bsp`, which is the file which Quake 3 loads.

Alternately, a different map altogether may be used, or a new one created. The map should be such that the player cannot die or become stuck in any part: for example, by falling into lava, drowning, or falling off the map (like in some of the Quake 3 “spacey” maps). The `srcvis.bat` file will need to be altered to specify the new map to be loaded.

Lastly, the `filenames` file may be edited to change the mapping of real source files to in-game `sourcefiles`. The filename at line n of the `filenames` file will be displayed by the sourcefile with `itemid` of n . (The minimum valid `itemid` is 1.)

This whole process is rather user unfriendly, but should be improved in the future, perhaps even incorporating an automatic layout of the source files, depending on the relationships between them.

8 Evaluation

8.1 Evaluation of the Source Code Comprehension Tool

The implemented tool met all the major functional requirements listed in the design section: source code files are represented as physical, movable 3D entities in a 3D world. They can be navigated via a web browser like interface which displays the files with syntax highlighting and hyperlinks. Users can see each other, and can customise their appearance to be unique using the normal Quake 3 model choosing system. A player can lock their view with another player, so that giving guided tours is possible. Users can “draw” on the source files, thereby indicating important parts of the code to other players also viewing that file (the “drawing” fades over time to reduce clutter). The chat functionality is not completely implemented yet, as it does not work correctly for users who are viewing contents of files, however it does work well in the normal 3D view.

There was no real usability trial done. However, during debugging, the help of several people was enlisted, and the following observations were made:

- ◆ Since everyone was in the same room, no-one bothered using the chat functionality.
- ◆ Sometimes in a place where there were lots of files, the file name labels would overlap, being difficult to read. One person came up with an interesting use of the Quake 3 ‘zoom’ function to overcome this. By holding down the ‘Ctrl’ key, the player’s view is magnified, showing less files, and separating the filenames labels more, making them easier to read.
- ◆ An unplanned feature was discovered: a player who is currently carrying a file can run up to another player, and drop the file onto him. The other player’s view will then show the contents of that dropped file. It is interesting to note that this feature arose by itself due to basing the tool on a 3D physical world metaphor. In the future, this sort of interaction could be used, for example, for a 3D interface to a source code version control system: a user walks up to a “vault,” where the checked-out files are dispensed, picks one up, carries it to their virtual workplace to work on it, and then carries it back and “drops” it into a chute leading back to the vault, to check-in the changes.

- ◆ In an early version of the tool, the lock view feature was only implemented within one source file view – as soon as the tour guide clicked on a link, the other participants’ views became unlocked, and they did not automatically follow the tour guide to the destination of the clicked link. The participants found this annoying, so the lock view feature was changed in the current version, to allow groups to follow the leader between files automatically. Unfortunately, this seems to have had the effect of turning the players who are being guided around into “zombies” – all that they do is look at the screen, and have no control of what they are looking at. This may lead them to losing attention quickly. Perhaps an intermediate solution could be implemented, where if the leader clicks on a link, that link is highlighted on the other player’s screens, but then they have to click on it. Whether this is a good solution should be investigated in the future.

8.2 Evaluation of Quake 3 Game Engine Suitability

Quake 3 was good choice for the underlying implementation platform. Since other game engines were not tried, no conclusion can be drawn about whether it was the best one available. There were some minor problems encountered with using this engine, listed below.

- ◆ There seems to be a bug in the game engine itself, whose source code is not currently available to the public. This bug manifests itself when a large number (more than 8000) of triangles are drawn from within the UI QVM. See Section 6.2.3.2 for details. This is an important example of how using the engine for what it was not designed for (displaying lots of text in the UI) may test the engine in ways that a normal game wouldn’t, revealing hidden bugs. Since Quake 3 should be open-sourced soon, this bug should be able to be fixed.
- ◆ Learning to modify the game engine took a lot of time, and was mostly done through trial and error. There is no documentation of the code provided by idSoftware. There are some basic third party resources [20,21], and these serve as a good introduction to the basics. However, it was often necessary to walk through the existing code by hand and figure out how it worked and how best to modify it.

- ◆ The game code which runs on the VMs is written in C. This has the usual side-effect of having the implementation of a particular entity split across many different files. This is not necessarily a bad thing in itself, but sometimes this leads to subtle bugs where some other part of the code in another file unexpectedly alters the state of an entity.
- ◆ The lack of dynamic memory allocation in the VMs is a limitation, but can be easily worked around. Either a large static array can be used and memory allocated out of that by hand (as was done in the `cgame.qvm` modification), or once the game engine source code is opened, this functionality may be added. However, such an addition would need to be carefully planned, since presumably there was a good reason for not including this functionality in the first place. Another way to work around this issue, which is not optimal, is to compile the game code as a win32 DLL. This can be loaded by Quake 3, and can make use of all native win32 functions such as dynamic memory allocation. However, this solution is non-portable, and removes the security barrier which is provided by executing the code in a QVM.
- ◆ Another result of the game engine code being, for the moment, closed, is that the network protocol is fixed. New character string messages can be added easily, but these need to be assembled at one end, and parsed at the other. In addition, these messages are not associated with any particular entity (other than, in the case of client-originating messages, which player sent them). This was a problem in the implementation of the tool since the new `itemid` field of `sourcefiles` had to be communicated from the server to the client. Fortunately, a field in the existing packet structure was found which was rarely used, and so was reused to transmit the `itemid`. While this worked for this particular tool, the available space is very limited, and may not be sufficient for other visualisation tools. A workaround based on sending the extra information via text messages and storing them in a look up table at the destination could be implemented, or, better, the network protocol could be made more flexible once access to the game engine source code is available.
- ◆ The last significant problem encountered was the limited capability of the UI system built into the game, specifically that it did not have a text scroll box element. However, all functions needed for implementing such an element were easily accessible, so there were no problems with coding this by hand (other than the mysterious 8000-triangle limit mentioned above.)

9 Conclusion

It is possible to use a game engine as the basis for an information visualisation tool, and thereby save a lot of implementation time by reusing the functionality already implemented in the game engine. The biggest benefit is obtained when as much of the game engine functionality is reused as possible, i.e. in 3D, interactive, multi-user visualisations based on a metaphor involving physical entities.

In order to simplify the implementation process, it is important to choose a visualisation which uses a metaphor of there being several distinct dynamic entities in a static 3D world that the users can interact with. There should typically be much less than 500 of these entities, but this limit depends on the game engine chosen, the amount of modifications needed, and the performance and hardware requirements of the result. There typically is also a limitation on the maximum allowed map (3D world) size, although this can be usually worked around by stitching together several smaller maps.

Out of the two ways of reusing a game engine, that is either writing a new set of source code which uses the engine, or modifying an existing game to customise it for producing the desired visualisation, the second is much easier and was therefore used in this project. There is still considerable work involved in customising an existing game, especially as in this case, where there was very little documentation of the code available.

The tool which was implemented in this project seems to work well as a source comprehension tool, this observation being based on an informal trial. It closely resembles browsing a cross-referenced source code base via a web browser, with the addition of multi-user capability and the ability to use one's spatial memory to remember the structure of the code.

The implemented tool may make a good basis for future software visualisation projects, wherein various graphical representations of information could be superimposed on top of the existing visualisation. If the user has already stored the basic structure of the source code files in his spatial memory, he could easily identify what parts of the source code the superimposed information related to.

10 Future Work

Minor improvements to the current tool:

- ◆ Create better models to display the sourcefiles, perhaps modify them dynamically to show some attributes of the files other than size (e.g.: age.)
- ◆ Improve the scrollbar so it can be dragged like a standard windows scrollbar.
- ◆ Reuse an AI route-finding algorithm in Quake 3 for telesliding.
- ◆ Improve the chat functionality, so that it works even in the file view.
- ◆ Create a 2D overview map, so can see what files and players are where.
- ◆ When a user holds their mouse over a link, display the destination file and line number. Perhaps display this on a superimposed 2D overview map.
- ◆ Only draw trails for players who are looking at same part of code.
- ◆ If a player is carrying a file, display its name above the player. Decide what should happen if a player clicks on a link that leads to a file which is currently being carried by someone – in the present implementation, the player will slide off to infinity.
- ◆ Improve the font used for file display – it seems to be stretched vertically and is hard to read.
- ◆ Add VOIP support so that players can talk to each other in voice via the internet.
- ◆ If a player picked up a source file, display its name in the player's view.
- ◆ Add search functions, to be able to search for text within a file or all files.
- ◆ Include ability to access doxygen generated documentation, not just the cross-referenced source code.
- ◆ Initially store all the source files on the server, and download them from there at game start or as needed – currently, every client needs to get a copy of the files by hand.
- ◆ Add the ability to save the location of source files after they have been moved, and load them at game start.
- ◆ Once Quake 3 is open-sourced, create new maps and textures (or use freely-licensed ones) so that the whole tool may be freely distributed.
- ◆ Add support for carrying multiple files, if this seems necessary.
- ◆ Improve dox2html as described in Section 6.1.
- ◆ Package the tool so as a proper Quake 3 Mod, and so that it can be loaded and unloaded via the in-game menus like any other mod, without needing a special batch script to start it.
- ◆ Test what happens when people arrive or leave part-way through a game session, some things like the lock view feature and the portrait drawing may fail – fix accordingly.

Major extensions to the tool:

- ◆ Automate the map generation, via an analysis of the source code, and with input from the user.
- ◆ Consider adding interactions with some software engineering tools, for example CVS (see Section 8.1).
- ◆ Consider superimposing other software visualisations on top of the visualisation generated by this tool (see Section 9).

Major future research questions:

- ◆ Investigate the suitability of game engines of other genres (RPG, RTS) for information visualisation.
- ◆ Investigate using this tool for visualising other repositories of hyperlinked documents – for example, using hyperlinks to show citations in a (small, due to the limit on the number of entities) digital library.
- ◆ Consider representing entities smaller than files in the tool – perhaps source files could be represented by a stack of function definitions, which could either be manipulated as a whole, or individually moved around. This could be used for refactoring programs.

11 References

1. Moloney, J, Amor, R., Furness, J., and Moores, B. Design Critique Inside a Multi-Player Game Engine, Proceedings of the CIB W78 Conference on IT in Construction, Auckland, New Zealand.
2. Manojlovich, J., Prasithsangaree, P., Hughes, S., Chen, J. and Lewis, M. UTSAF: A Multi-Agent-Based Framework for Supporting Military-Based Distributed Interactive Simulations in 3D Virtual Environments, Proceedings of the 2003 Winter Simulation Conference, New Orleans, LA.
3. Herwig, A., Paar, P. Game Engines: Tools for Landscape Visualization and Planning?, Trends in GIS and Virtualization in Environmental Planning and Design, 2002.
4. Chao, D., Doom as an Interface for Process Management, SIGCHI'01, Seattle, WA, USA.
5. Heckenberg, S.G., Herbert, R.D. and Webber, R. (2004). Visualisation of the Minority Game Using a Mod. In Proc. Australasian Symposium on Information Visualisation, (invis.au'04), Christchurch, New Zealand. Conferences in Research and Practice in Information Technology, 35. Churcher, N. and Churcher, C., Eds., ACS. 157-163.
6. Age of Empires II, www.microsoft.com/games/age2/
7. World of Warcraft, www.worldofwarcraft.com
8. OGRE Object-oriented Graphics Rendering Engine, www.ogre3d.org
9. Crystal Space 3D, crystal.sourceforge.net
10. Irrlicht Engine - A free open source 3d engine, irrlicht.sourceforge.net
11. GarageGames, www.garagegames.com
12. feshmeat.net project reviews – Irrlicht, freshmeat.net/articles/view/1182/irrlichht

13. The Nebula Device 2, nebuladevice.cubik.org
14. OpenSceneGraph, www.openscenegraph.org
15. OpenSG Home, www.opensg.org
16. id Software, www.idsoftware.com
17. valve corporation, www.valvesoftware.com
18. Epic Games, www.epicgames.com
19. John Carmack's Blog, www.armadilloaerospace.com/n.x/johnc/Recent%20Updates
20. Quake III: Arena, baseq3 mod commentary, www.icculus.org/~phaethon/q3mc/q3mc.html
21. Code3Arena www.planetquake.com/code3arena/
22. Gagnon, D. Videogames and Spatial Skills: An Exploratory Study. *Educational Communication and Technology* 33, 4, 263-275, 1985.
23. Vicente, K., Hayes, B. and Williges, R. Assaying and Isolating Individual Differences in Searching a Hierarchical File System. *Human Factors* 29, 3, 349-359, 1987.
24. Leitheiser, B. and Munro, D. An Experimental Study of the Relationship Between Spatial Ability and the Learning of a Graphical User Interface. In *Proceedings of the Inaugural Americas Conference on Information Systems*, 1995.
25. Cockburn, Andy & McKenzie, Bruce, Evaluating the Effectiveness of Spatial Memory in 2D and 3D Physical and Virtual Environments. *Spatial Cognition*, Apr2002, Volume No. 4, Issue No. 1, p203-210.
26. Andrian Marcus, Louis Feng, Jonathan I. Maletic, 3D representations for software visualization. *Proc. 2003 ACM Symp on Software Visualization*
27. Ball, T. and Eick, Software Visualization in the Large. *Computer*, vol. 29, no. 4, April, pp. 33-43.
28. Loe M. G. Feijs, Roel de Jong, 3D Visualization of Software Architectures. *Commun. ACM* 41(12): 72-78 (1998)
29. Thomas Panas, Rebecca Berrigan, John Grundy, A 3D Metaphor for Software Production Visualization, *Seventh International Conference on Information Visualization (IV'03)* London, England July 16 - 18, 2003 p. 314
30. Doxygen, www.doxygen.org
31. PlanetQuake Quake 3 Mod Making Forum, www.forumplanet.com/planetquake/forum.asp?fid=2259
32. The gzip homepage, www.gzip.org

12 Acknowledgements:

I would like to thank:

- ◆ The Faculty of Science of the University of Auckland for the funding for this project.
- ◆ Burkhard Wuensche, John Hosking and John Grundy for their supervision and guidance.
- ◆ Burkhard Wuensche for the idea on which this project is based.
- ◆ idSoftware for Quake 3.
- ◆ The Code3Arena team and phaethon for their helpful articles about Quake 3 Modding.
- ◆ The PlanetQuake Quake 3 Modding forum community, especially AnthonyJ and ^misanthropia^ for their helpful and prompt replies.
- ◆ The doxygen team.
- ◆ The gzip team for the sample code to use in the demo.